Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Recuperação Automática da Modelação Comportamental com Aplicações ao Ensaio Baseado em Modelos

Filipa Pires Duarte da Silva
(Licenciada)

Dissertação apresentada para a obtenção do
Grau de Mestre em Engenharia Informática.

Julho 2008

Esta dissertação foi preparada sob a orientação do
Professor Doutor Fernando Brito e Abreu
do Departamento de Informática
da Faculdade de Ciências e Tecnologia
da Universidade Nova de Lisboa.

Resumo

O problema dos sistemas de software legados sem documentação ou com documentação obsoleta, continua a ser uma realidade no mundo empresarial. O progressivo aumento da dimensão e complexidade dos sistemas desenvolvidos vem aumentar a necessidade de existirem mecanismos de modelação e documentação de apoio às actividades de manutenção e teste. Apesar da investigação que tem sido levada a cabo para tentar apresentar cada vez melhores abordagens à resolução deste problema, o seu uso no mundo empresarial ainda é muito esparso. Tal deve-se, pelo menos em parte, ao facto de muitas das abordagens propostas acabarem por representar um acrescento do esforço e do tempo, que as organizações não estão em condições de disponibilizar.

Esta dissertação contribui com uma abordagem automatizada de suporte às actividades de documentação de sistemas, de manutenção e de testes. Para demonstrar a aplicabilidade e usabilidade da abordagem, propõe-se a implementação de uma ferramenta de apoio. A abordagem proposta baseia-se em técnicas já existentes e consolidadas, mas propõe extensões que permitem melhorar a sua integração, usabilidade e eficiência na sua aplicação. O problema fulcral aqui tratado é a inexistência ou insuficiência de documentação sobre os sistemas desenvolvidos. De modo a mitigar este problema, é apresentado um mecanismo de recuperação da modelação dinâmica de sistemas legados e respectiva geração de artefactos documentais, nomeadamente diagramas de sequência (UML), cartões CRC e matrizes de CRUD. Finalmente, introduzem-se técnicas de rastreabilidade e de apoio a testes de qualidade e cobertura para os sistemas construídos, usando uma metáfora de coloração de diagramas UML.

Palavras-Chave:

UML; Engenharia Inversa; Diagramas de sequência; Cartões CRC; Matrizes CRUD; Análise de Impacto; Java; XMI.

Abstract

Legacy software systems with inadequate or no documentation are a major problem to many organizations. The increasing size and complexity of software applications developed nowadays have raised the need of create modelling and documenting techniques to support maintenance and testing activities. In spite of the research to find suitable approaches to solve this problem, the technology transfer of such approaches to the software industry remains scarce. One of the main shortcomings of many of the current approaches is that the associated effort and time consumption represent an overwhelming overhead that organizations cannot cope with.

This dissertation contributes with an automatic approach to support documenting, maintenance, and testing activities. To validate its applicability and usability, we propose the implementation of a supporting tool prototype. The proposed approach is based in existing and established techniques, which are extended here to improve their usability and efficiency. The key problem we are addressing is the insufficient or inexistent documentation of developed systems. In order to mitigate this problem, we present a technique for (dynamic) model recovery from legacy systems. The proposed technique allows generating document artefacts, such as UML diagrams, CRC Cards, and CRUD matrices. Finally, we introduce techniques to support traceability and quality assurance activities (namely coverage analysis), using a colouring metaphor upon several UML diagrams.

Keywords:

UML; Reverse Engineering; Sequence Diagrams; CRC Cards; CRUD Matrices; Impact Analysis; Java; XMI.

Índice

Índice de Figuras

Capítulo 1

Introdução

Conteúdo

Este capítulo introduz os conceitos principais desta dissertação, descreve uma motivação para a recuperação automática de artefactos documentais de sistemas legados e enumera os principais objectivos e contribuições. No final, faz uma apresentação sumária dos conteúdos presentes nos restantes capítulos do documento.

1 Introdução

1.1 Introdução Geral

Foram feitas algumas tentativas para prever a evolução dos sistemas de informação, como a realizada por Bill Gates em 1985 [Wikipedia, 2008], em que o autor previu que iria existir um computador por secretária em cada escritório e possivelmente em cada casa e que cada um iria conter software da Microsoft. Segundo o mesmo autor, é espantoso como os computadores evoluíram nas duas últimas décadas [Gates, 2001]. O software tornou-se uma tecnologia indispensável a muitas áreas, como a banca, a educação, a ciência, etc, e até permitiu criar ou estender novas tecnologias, como a engenharia genética ou as telecomunicações. A sua influência na vida de cada um tem sido tão notória que temos observado a alteração e criação de novos hábitos sociais e de consumo.

"Today, computer software is the single most important technology on the world stage." [Pressman, 2005]

A situação descrita levou a um aumento da produção e proliferação de sistemas de software. Do mesmo modo, os sistemas já existentes têm de ser continuamente modificados e adaptados às novas necessidades e requisitos. Este fenómeno tem vindo a preocupar e a captar as atenções nos últimos 40 anos.

"Legacy software systems…were developed decades ago and have been continually modified to meet changes in business requirements and computing platforms. The proliferation of such systems is causing headaches for large organizations who find them costly to maintain and risky to evolve." (Dayani99 in [Pressman, 2005]).

No entanto, os sistemas legados não se limitam a sistemas antigos. Sistemas desenvolvidos recentemente podem originar um sistema legado, se apresentarem algumas das características que os definem. Exemplos destas últimas são, a falta de

qualidade, documentação inexistente ou inadequada, código não estruturado, complexo e confuso, desenho não extensível ou ausência de um historial de gestão de alterações.

Muito trabalho tem sido desenvolvido para mitigar esta situação, como por exemplo a definição de linguagens padronizadas de modelação – *Unified Modelling Language* (UML) [OMG, 2007] e de processos de desenvolvimento (SPICE (Dorling91 em [Silva and Videira, 2005]), CMMI [CMUniversity, 2008]). Por outro lado, também muito se tem discutido sobre técnicas de recuperação de qualidade de sistemas legados. É neste âmbito que se enquadra esta dissertação.

1.2 Motivação

Durante décadas temos vindo a observar inúmeros problemas caracterizados como fazendo parte da crise de software [Gibbs, 1994]. Os sistemas legados são em grande parte aqueles em que essa "crise" se manifesta. Com o aumento da dimensão e complexidade dos sistemas desenvolvidos actualmente, veio intensificar-se a necessidade de criar mecanismos de apoio à modelação, documentação, manutenção e teste dos mesmos. Se pensarmos na rotatividade das equipas de desenvolvimento ou nos requisitos exigentes que são pedidos, podemos estimar que o processo de criação de software pode ser uma tarefa difícil sem o apoio desses mesmos mecanismos. Apesar do trabalho realizado, por vários investigadores, na tentativa de apresentar cada vez melhores abordagens e técnicas à resolução e apoio deste problema, por vezes verifica-se que um número considerável de empresas constrói os seus sistemas sem o uso directo dos referidos mecanismos [Hvam, Jesper et al., 2003].

Um contributo importante trazido há cerca de uma década foi a definição de uma linguagem padronizada de modelação, a UML (*Unified Modelling Language*). A UML é uma linguagem que utiliza uma notação (gráfica) padrão para especificar, construir, visualizar e documentar sistemas [Nunes and O'Neill, 2004], mais precisamente para criar modelos abstractos.

A UML, em particular a versão 2.0, propõe vários diagramas organizados em três visões principais, em que cada uma representa uma perspectiva diferente do sistema, permitindo diferentes graus de abstracção. Em baixo é apresentada uma tabela resumida, dos vários tipos de diagramas agrupados pelas visões que descrevem.

Visão Funcional	Modelo de Requisitos	Diagramas de casos de uso
	Modelo Funcional	Diagramas de actividades
Visão Estrutural	Modelo Estrutura	Diagramas de Pacotes
		Diagramas de Classes
		Diagramas de Objectos
		Diagramas de Estrutura composta
	Modelo Arquitectural	Diagramas de componentes
		Diagramas de instalação
Visão Dinâmica	Modelo Comportamental	Diagramas de Sequência
		Diagramas de Comunicação
		Diagramas Temporais
		Diagramas de Estados
		Diagramas de Visão Geral da Interacção

Figura 1. Diagramas da UML.

A visão funcional representa os requisitos funcionais do sistema do ponto de vista do utilizador. A visão estática ou estrutural representa a estrutura estática do sistema usando classes, objectos, atributos, operações e relações. A visão dinâmica representa o comportamento dinâmico do sistema, mostrando as interacções entre os objectos e as alterações ao estado de cada um.

A UML disponibiliza ainda uma representação padronizada de interoperabilidade entre ferramentas de modelação através do formato XMI (*XML Metadata Interchange*) [OMG, 2008].

Apesar da UML representar uma colecção das melhores práticas de engenharia que têm provado sucesso na modelação de sistemas, principalmente sistemas complexos e de grandes dimensões, na indústria apenas alguns dos seus diagramas são realmente realizados. A partir de [Gouveia, 2008], podemos concluir que os diagramas com maior aceitação são os diagramas de casos de utilização e os diagramas de classes, e mesmo assim, ficam muitas vezes obsoletos com as alterações que são feitas aos requisitos. Assim sendo, a parte comportamental fica inevitavelmente “esquecida”. Os diagramas de casos de uso, e respectivas descrições de cenários, apresentam os requisitos do

sistema do ponto de vista do utilizador, visão do domínio do problema. Por sua vez, os diagramas de classes apresentam a estrutura do sistema, permitindo uma visão do domínio da solução. Os diagramas de sequência vêm fazer a ligação entre estas duas visões, porque descrevem os cenários dos casos de uso, através de objectos e das mensagens trocadas entre si. Se este diagrama não for construído, fica a faltar a ligação entre os dois diagramas anteriores, perdendo-se a rastreabilidade entre o domínio do problema e o da solução (ver Figura 2).

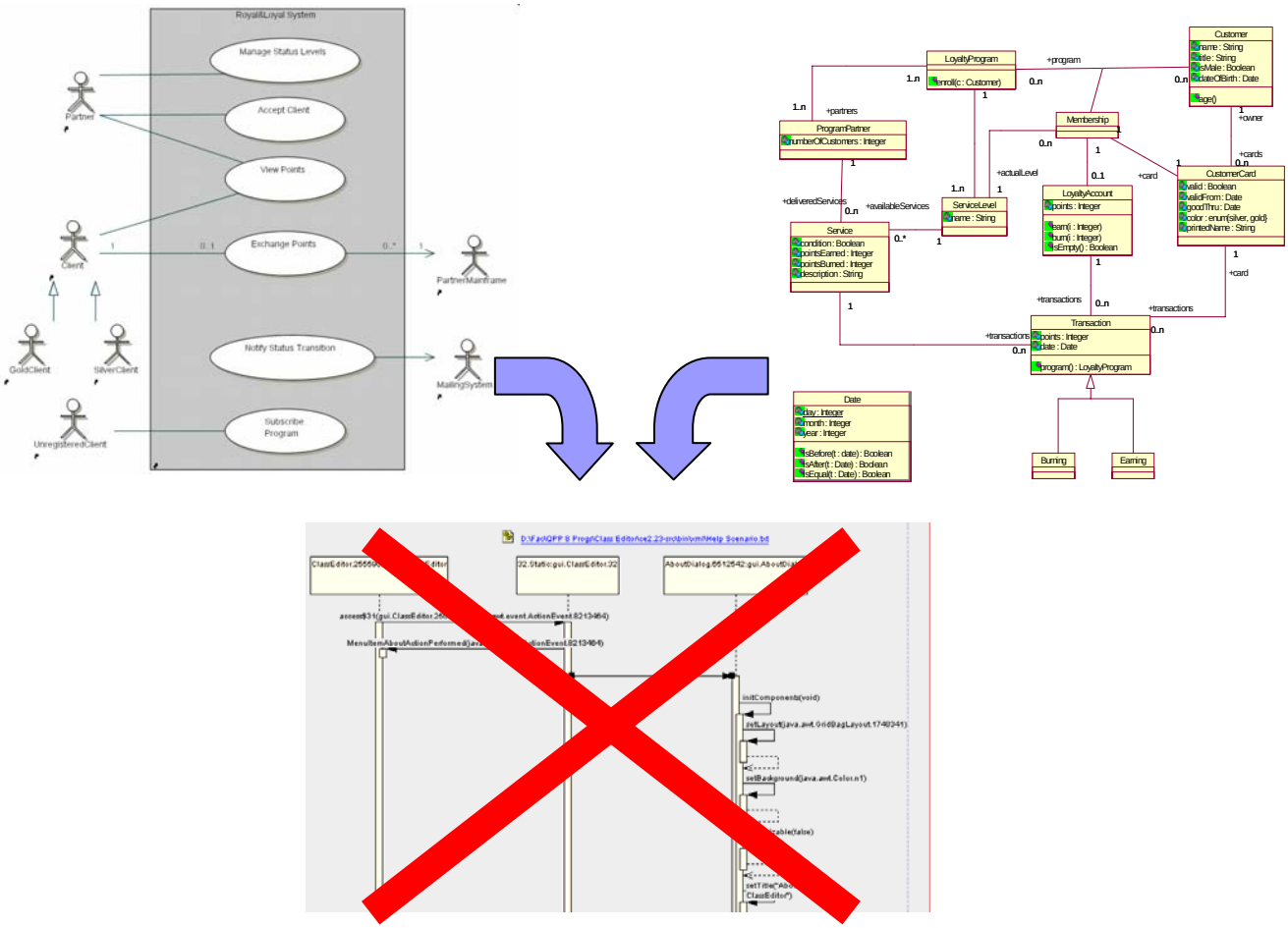


Figura 2. Diagrama casos de uso (à esquerda), Diagrama de classes (à direita) e diagrama de sequência (em baixo). A figura descreve a perda de rastreabilidade nos diagramas UML.

Para além da engenharia de software, também a engenharia inversa (*reverse engineering*) se tem desenvolvido, para procurar formas de recuperar informação dos sistemas legados de modo a poder documentá-los.

A engenharia inversa é o processo de analisar um sistema implementado e conseguir extrair a arquitectura do mesmo. O resultado da implementação (código fonte) é processado de modo a criar os artefactos que traduzam as decisões normalmente tomadas na fase de desenho, numa inversão do ciclo de vida natural do software. Na engenharia inversa, os requisitos e o desenho devem ser capturados do sistema legado,

através da extracção de informação. O processo deve ser capaz de gerar visões distintas do sistema criando modelos ou outros artefactos para as representar, recuperar (ou criar) a documentação perdida. Esses resultados são muitas vezes usados para detectar efeitos secundários a alterações propostas (análise de impacto).

A engenharia inversa é o processo de examinar sem modificar, não envolvendo qualquer alteração ao sistema ou a criação de um novo sistema.

Em 1995, os autores de [Quilici, 1995] questionavam-se sobre a possibilidade de derivar automaticamente as especificações necessárias, de um modo eficiente ao nível dos custos e do tempo dispendido. Nessa altura, a resposta era negativa e os autores afirmavam que o estado de arte corrente ainda estava longe do seu objectivo. Desde essa altura que a investigação tem evoluído e já não se encontra tão longe assim do seu objectivo [ver capítulo 6]: as técnicas de engenharia inversa têm evoluído no sentido de automatizar cada vez mais esse processo.

Construir todos os diagramas para um sistema e principalmente mantê-los não é uma tarefa fácil. A minha experiência no ensino, leva-me a pensar que construir alguns dos diagramas do UML pode representar um grande consumo de tempo, custos e recursos, que muitas vezes desmotiva a sua realização. A recuperação estática da estrutura dos sistemas, nomeadamente recuperação de diagramas de classes, é possível encontrar em muitas ferramentas da UML. E sendo assim, alguns autores identificam como o passo seguinte, a recuperação do comportamento do sistema [Rountev, Volgin et al., 2005]. Sem esta visão dinâmica do sistema, perde-se a ligação entre os requisitos e a arquitectura do sistema, ou a compreensão do funcionamento do mesmo. No entanto, é muito difícil, e às vezes mesmo impossível, sem avaliarmos a execução dos sistemas, tirar informações conclusivas sobre o seu comportamento real [Briand, Labiche et al., 2003].

1.3 Objectivos da Dissertação

Nesta dissertação apresenta-se uma contribuição para mitigar os problemas atrás referidos. É proposta uma nova técnica de geração automática e integração de artefactos que facilitam a compreensão de programas complexos, o seu ensaio e o planeamento da sua evolução (através da análise de impacto das alterações propostas ao nível dos requisitos). A inovação não está nos artefactos *per se*, dado já serem conhecidos há bastante tempo, mas no modo como eles se conseguem obter e na sua integração.

A técnica aqui descrita é parte de um trabalho integrado, que tem sido desenvolvido em paralelo [Gouveia, 2008], suportado por uma ferramenta denominada por *ReModeler*. O processo proposto nesta dissertação, começa com uma captura do sistema a analisar. Esta captura é dinâmica, ou seja, levada a cabo durante a execução do mesmo, e é realizada para cada cenário de utilização do sistema, individualmente. Ela permite recolher informações sobre as interacções entre objectos e sobre que partes dos sistemas são executadas durante cada cenário. A captura é possível através da instrumentação do código fonte, tirando partido da funcionalidade de *weaving* da tecnologia dos aspectos.

Após a captura propõe-se manipular a informação recolhida de forma a gerar vários artefactos, tais como diagramas de sequência temporizados, cartões CRC estendidos, matrizes CRUD de implementação de requisitos e diagramas UML coloridos (ver capitulo 2).

A principal contribuição que se espera oferecer com esta dissertação é a capacidade de gerar automaticamente, de uma forma e num contexto inovador, artefactos a partir de sistemas legados.

A ideia é, através das facilidades introduzidas pela captura dinâmica, recolher a informação necessária à representação comportamental do sistema, mas também para representar a estrutura das classes e suas relações.

O essencial que se pretende é garantir rastreabilidade entre os modelos e o código, através de um conjunto de abstracções como cartões CRC, matrizes CRUD ou diagramas coloridos.

Sumariamente, os objectivos que se espera alcançar com esta dissertação são:

- Permitir a realização de análise de impacto através dos artefactos, como cartões CRC e matrizes CRUD, que indicam o modo como as classes estão representadas entre si e como colaboram, e como, a esse nível, os requisitos estão implementados.
- Reduzir a dificuldade em conseguir os artefactos documentais do sistema. O modo de geração apresentado, com reduzidos custos de tempo e esforço, vem colmatar o problema da difícil criação dos artefactos que leva muitas empresas a deixar estas actividades para segundo plano, originando novos sistemas legados. Para além disso, as novas características propostas, aumentam a simplicidade de análise e compreensão dos artefactos propostos.

- Suportar a actividade de ensaio, gerando análises de cobertura e intensidade de utilização ao nível dos modelos, e gerando relatórios estatísticos com informação útil para essas actividades, como os recursos usados (memória, CPU, etc) ou o número de invocações de métodos, etc.
- Diminuir o fosso que existe entre a modelação estática e dinâmica, que leva a perdas de rastreabilidade, através da recuperação dos diagramas de sequência. Os diagramas coloridos gerados no âmbito dos testes de cobertura são um bom exemplo da união dos dois "mundos".

Em suma, espera-se que esta dissertação ajude a guiar o processo de recuperação e entendimento de sistemas legados, minimizando o tempo e o esforço habitualmente necessário para o conseguir.

1.4 Organização da dissertação

Para além do presente capítulo introdutório, esta dissertação é composta por um conjunto de seis capítulos.

- Capítulo 2: Neste capítulo é relatado detalhadamente o problema que motivou esta dissertação. É também descrita a solução que se pretende atingir, sob a visão do domínio do problema, identificando não só a definição de todo o processo proposto, mas também os requisitos necessários à sua implementação.
- Capítulo 3: Este capítulo descreve a arquitectura da solução proposta através dos seus componentes, detalhando cada um deles ao nível da sua implementação por pacotes.
- Capítulo 4: Neste capítulo é apresentado o desenho detalhado para a solução proposta, descrevendo os aspectos relacionados com a implementação do *ReModeler*. Aqui são descritos pormenorizadamente os pacotes que constituem a ferramenta construída, tendo sido incluídos alguns detalhes da implementação.
- Capítulo 5: Neste capítulo o processo, suportado pela ferramenta desenvolvida, é validado pela aplicação a um caso de estudo. Todos os passos dessa aplicação são descritos, identificando claramente os *inputs* e *outputs* obtidos.

- Capítulo 6: Este capítulo fornece uma panorâmica sobre o trabalho que tem sido desenvolvido nas áreas envolventes a esta dissertação. O estudo apresentado neste capítulo está organizado pelos vários artefactos gerados no processo proposto.
- Capítulo 7: Neste capítulo são apresentadas as conclusões para esta dissertação, identificando as linhas de orientação gerais para a continuação futura do trabalho desenvolvido.
- Anexo A: Neste anexo são apresentadas e justificadas as tecnologias utilizadas no desenvolvimento da ferramenta *ReModeler*.
- Anexo B: Este anexo descreve algumas ferramentas disponíveis no mercado que implementam técnicas de *reverse engineering* relacionadas com as produzidas nesta dissertação.

Capítulo 2

Especificação de requisitos

Conteúdo

Neste capítulo é relatado detalhadamente o problema que motivou esta dissertação. É também descrita a solução que se pretende atingir, sob a visão do domínio do problema, identificando não só a definição de todo o processo proposto, mas também os requisitos necessários à sua implementação.

2 Especificação de requisitos

2.1 Introdução Geral

Nesta dissertação tenta-se mitigar problemas encontrados que afectam a qualidade do desenvolvimento de software ou mesmo do produto já desenvolvido. Num sistema de software, os requisitos estão muitas vezes dispersos no código, espalhados por vários módulos. Mesmo que se seja o programador original, é frequente desesperar-se com este cenário, principalmente para sistemas grandes, complexos e em funcionamento. Esta mesma situação ocorreu no desenvolvimento da ferramenta desenvolvida para suportar as propostas contidas nesta dissertação. Ao fim de algumas semanas de programação e com cerca de quatro dezenas de classes implementadas, tornou-se extremamente difícil identificar claramente que efeitos uma alteração de um requisito poderia ter para o resto do sistema.

O foco desta dissertação é a geração automática de documentação para sistemas legados e o auxílio no desenvolvimento e evolução de novos sistemas, suportando actividades de manutenção, testes e gestão de impacto de alterações.

A automatização reduz custos e aumenta a fiabilidade de resultados. Por esta razão é possível afirmar que os conceitos presentes nesta dissertação se enquadram na área de Automatização da Engenharia de Software (*Automated Software Engineerng – ASE*) [Grünbacher and Ledru, 2004]. A ASE aplica a computação a actividades da Engenharia de Software, para permitir o desenvolvimento e evolução de sistemas, principalmente de grandes dimensões e complexos, de uma forma económica e dentro do tempo previsto. Isto inclui o estudo de técnicas de construção, compreensão, adaptação e modelação de artefactos e processos de software. Automatizar melhora a fiabilidade por reduzir a possibilidade de erros humanos, aumenta a eficiência porque as tarefas automatizadas são normalmente mais rápidas que as manuais e incrementa a probabilidade de sucesso no processo de desenvolvimento [Kitchens, 2006].

Esta dissertação propõe uma nova técnica de geração automática e integração de um conjunto de artefactos que facilitam a compreensão de programas complexos, o seu ensaio e planeamento da sua evolução (através da análise de impacto das alterações propostas ao nível dos requisitos). Os artefactos gerados visam mitigar alguns dos problemas inicialmente identificados, como o hiato do UML, na

separação e inexistência prática, da ligação entre a visão dos requisitos e a visão estrutural.

Existem vários tipos de intervenientes no processo aqui descrito, uns que se comportam como actores e outros que são os principais beneficiadores dos artefactos gerados. Podemos destacar os seguintes:

- membros das Equipas de Utilizadores
- membros das Equipas de Requisitos e Testes
- membros das Equipas de Desenvolvimento
- responsáveis pela gestão da qualidade dos sistemas
- responsáveis pela gestão de projectos de desenvolvimento de aplicações
- responsáveis pela gestão de alterações/ configurações

As principais contribuições previstas de cada artefacto produzido para cada um dos actores ao longo do ciclo de vida do software estão sumariadas na Tabela 1.

	Diagramas de Sequência temporais	Diagramas de coloridos	Cartões CRC	Matrizes CRUD
Cliente				
Utilizador final				
Testador	√	√		
Help desk (1ª linha de manutenção)				
Mecânico de software (2ª linha de manutenção)	√	√	√	√
Programador	√	√	√	√
Gestor de projecto			√	√
CIO				
Perito do domínio				

Tabela 1. Contribuições de cada artefacto para cada actor

Enquadrado no novo conceito de Modelação Aumentada (*Aumented Modeling*) [Abreu, Silva et al., 2007], este processo, implementado numa ferramenta denominada de *ReModeler*, visa em primeiro lugar o incremento da adopção do desenvolvimento suportado por modelos. Acredita-se que esta abordagem possa trazer grandes melhorias e vantagens para o processo de desenvolvimento de software e seus intervenientes. A primeira melhoria prevista é a da compreensão do impacto das alterações, pois é sabido que uma alteração num requisito pode provocar efeitos colaterais noutros requisitos e assim sendo é de grande importância para os membros das equipas de requisitos e testes que existam mecanismos de percepção desse impacto. Outra melhoria prevista é no planeamento (esforço / custos). São em geral conhecidos os requisitos que requerem alterações na sua implementação, bem como os recursos disponíveis para o efectuar, que são sempre escassos. No entanto é normalmente difícil estimar o esforço necessário para a sua concretização e teste. Embora não seja aqui realizado um modelo de estimação de custos/esforço, alguns dos artefactos propostos podem dar uma aproximação inicial às dependências geradas pela alteração de um requisito. Para os responsáveis pela gestão de projecto, para os membros das equipas de requisitos e testes e até para os membros das equipas de desenvolvimento, torna-se vital a existência de artefactos que auxiliem a identificação da dificuldade da alteração. Se for possível ter noção da maneira como um determinado requisito se relaciona com os restantes e como é que é implementado, torna-se mais fácil a identificação do esforço para o alterar. Sempre que se for proceder a uma alteração no desenho ou no código, por exemplo para corrigir um defeito relacionado com a eficiência, ou genericamente para melhorar a qualidade do código, será possível visualizar que requisitos poderão ser afectados por essa alteração. Por ultimo, prevê-se melhorar a análise de cobertura, através da disponibilização de representações diagramáticas contendo informação de cobertura da aplicação, o que se traduz numa mais valia para a equipa de verificação e validação e para os membros das equipas de engenharia de requisitos.

De entre os vários artefactos gerados pelo *ReModeler*, podem ser destacados quatro, que são descritos de seguida.

2.2 Diagramas de Sequência Temporizados

A modelação dinâmica de um sistema é fundamental para dominar a sua complexidade e compreender as suas particularidades. Os diagramas de interacção, dos quais os diagramas de sequência são um caso particular, são utilizados no UML para modelar os aspectos dinâmicos do sistema em termos dos objectos e ordem cronológica das suas interacções. Os diagramas de interacção permitem definir e clarificar a colaboração entre as classes do sistema. Normalmente são utilizados para ilustrar o comportamento do sistema num cenário de concretização de um caso de utilização.

Nos diagramas de sequência (ver Figura 3) representam-se os vários objectos que intervêm na execução, com uma linha temporal que acompanha o ciclo de vida de cada um. Sempre que um objecto envia uma mensagem a outros objectos fá-lo a partir da sua linha temporal, e quando tem controlo para o fazer. As mensagens, por sua vez, podem ser assíncronas ou síncronas, podendo representar várias acções, como a criação de novos objectos ou a invocação de mensagens. O UML 2.0 introduz ainda mecanismos para representar iterações, condições e várias fiadas de execução (*threads*).

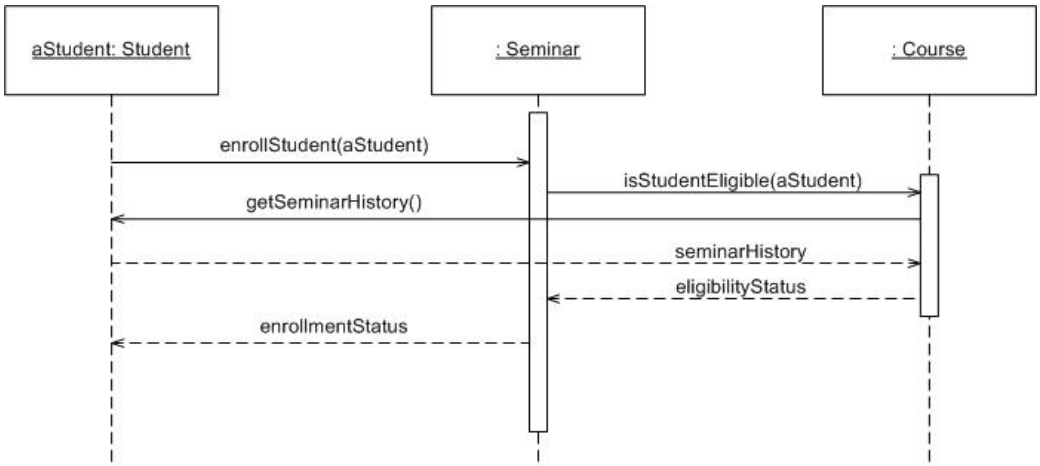


Figura 3. Diagrama de sequência da UML

Reforçando o que foi escrito no capítulo introdutório, estes diagramas, embora importantes para a compreensão de um sistema, são muitas vezes esquecidos. Isto acontece em parte pela dificuldade da sua criação, mas quando se trata da recuperação da documentação de um sistema legado, este problema torna-se ainda mais presente. É muito difícil, e às vezes até impossível de saber (devido à polimorfia), analisando apenas o código fonte, que métodos vão ser executados e que

objectos vão ser criados numa execução. Torna-se assim complicado seguir a execução do programa e por conseguinte criar um diagrama de sequência. Existem técnicas para a identificação da sequência de invocações de métodos, mas estas são de difícil aplicação a sistemas grandes e complexos [Briand, Labiche et al., 2003]. Nestes casos, torna-se indispensável a execução do sistema e respectiva monitorização da execução para capturar a informação necessária para a recuperação de modelos dinâmicos, como os diagramas de sequência da UML.

Um diagrama de sequência dá uma visão esquematizada e sequencial das trocas de mensagens, permitindo o melhor entendimento da implementação dos requisitos, através da visualização das acções internas. Em caso de erro ou acção inesperada, é possível percepcionar qual a causa. Por outro lado, é possível validar face aos requisitos e à estrutura do sistema, a sua conformidade na implementação, ou seja, o modo como os casos de uso, descritos na visão do domínio, estão implementados através das relações entre as várias classes, presentes na estrutura do sistema. Por estas razões, estes diagramas são úteis quer a programadores, quer a testadores e a mecânicos de software (os que fazem a manutenção do software).

Acresce que os diagramas propostos estendem os diagramas habituais, na medida em que adicionam características que visam melhorar a compreensão de quem os analisa. Em primeiro lugar, os diagramas apresentam um *link* (ver Figura 4) para a descrição textual do cenário a que correspondem, o que permite a leitura da descrição em língua natural associada a cada passo. Para além disso, cada mensagem representada tem adicionada uma marca temporal, correspondente ao instante temporal em que ocorreu, que vai permitir identificar a fronteira entre os vários passos e permitir a sincronização com outros elementos [Gouveia, 2008].

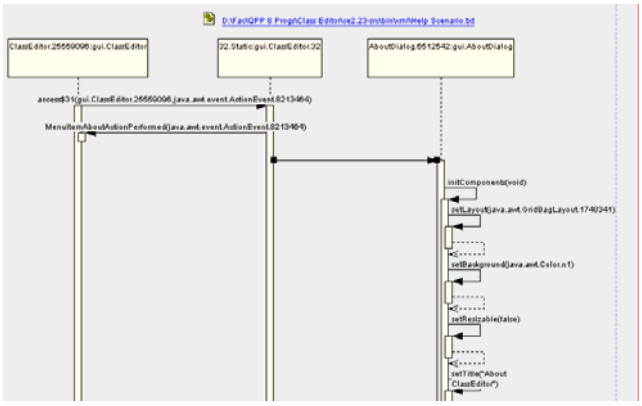


Figura 4. Diagrama de sequência estendido.

Os diagramas de sequência propostos podem ser clarificados através de diagramas de classes que constituem uma ontologia do domínio. Observando a Figura 5 é possível perceber que cada diagrama de sequência temporizado gerado representa um cenário de um caso de utilização, descrito por um conjunto de passos. Por sua vez, consegue-se definir os diagramas propostos como um caso particular de um diagrama de sequência UML (ver Figura 6 – à direita), que descreve as mensagens trocadas entre objectos do sistema (ver Figura 6 – à esquerda).

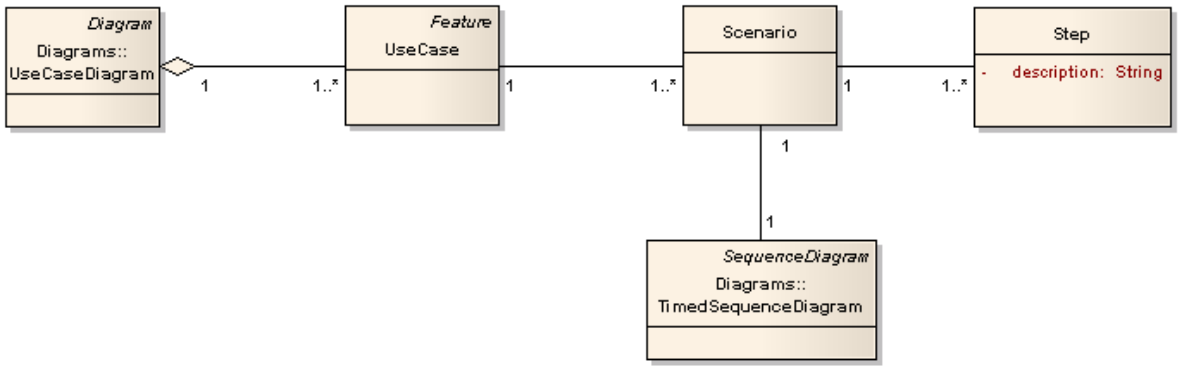


Figura 5. Especificação de Requisitos.

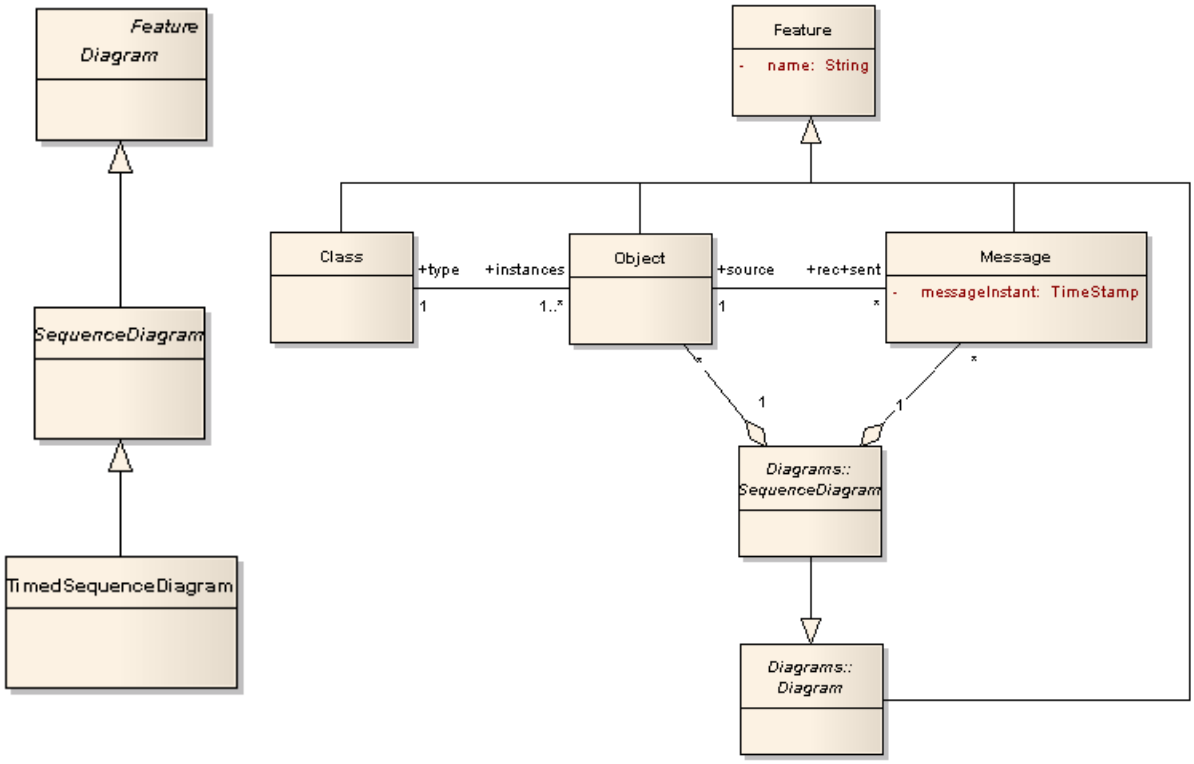


Figura 6. (À direita) Diagramas. (À esquerda) Representação dos componentes base.

2.3 Matriz de CRUD Estendida

Uma matriz CRUD é uma tabela que mostra as ligações entre processos e dados ou recursos. Essas relações podem ser caracterizadas pelo seu tipo: de criação (*Create*), de leitura (*Read*), de escrita (*Update*) ou de destruição (*Delete*), dos dados ou recursos.

As matrizes CRUD foram inicialmente propostas para modelação de dados no contexto das bases de dados relacionais [Kilov, 1990]. A matriz era criada contendo nas linhas entidades ou tabelas da base de dados e nas colunas funções, procedimentos ou módulos de um programa em que as entidades eram acedidas. Em cada célula eram depois colocadas as letras que correspondiam aos diferentes tipos de acesso, que em SQL são: INSERT - C, SELECT - R, UPDATE – U e DELETE – D (Create, Read, Update, Delete), tal como apresentado na Figura 7.

Tabelas vs Forms	Vendedor	Vinho	Marca	Variedades	Loja	Empresa
Cliente					CRU	CRU
Vinho		CRU	CRU	CRUD		
Marca			CRU			
Farinha				CRUD		
Vendedor	CRU					

Figura 7. Exemplo de uma Matriz de CRUD

O uso destas matrizes tem evoluído e podemos vê-las actualmente a serem usadas em vários contextos. Elas são usadas para mapear interfaces com as possíveis interacções do utilizador [Wikipedia, 2008], mas também para estruturar processos de negócio [Wang, Xu et al., 2005], entre outros.

Estas matrizes são criadas normalmente na fase de análise e desenho, na identificação e estruturação das responsabilidades dos processos e dados. No entanto elas são bastante úteis noutras fases e com outras finalidades. Na análise da qualidade e completude do modelo de dados, face aos requisitos que o sistema deve suportar, é comum representarem-se os resultados sob a forma desta matriz, para facilitar o encontro de falhas no modelo de dados.

Por outro lado, a sua estrutura permite analisar que tabelas ou entidades são usadas, e de que forma, facilitando a identificação de fontes de problemas de performance, como *bottlenecks* [Answers, 2001].

A sua forma de compactação e representação das estruturas e suas ligações pode ajudar a guiar a construção de baterias de testes para sistemas de grande dimensão e a manter esses mesmos sistemas. Quando é necessário efectuar uma alteração em alguma parte do sistema, é possível ter uma melhor percepção de que outras áreas necessitam de ser testadas também. Por estas razões, para os programadores e arquitectos de software estas matrizes representam uma forma de compreenderem como cada requisito foi implementado, disponibilizando um primeiro nível de avaliação de análise de impacto. Para auxiliar as tarefas de quem faz a manutenção do sistema, as matrizes podem apresentar informação adicional quantitativa, nomeadamente mostrando o número de vezes que cada categoria nominal ocorreu por classe ou por cenário, baseado no número de invocações de métodos.

Recentemente, a técnica das matrizes CRUD começou a estender-se à análise e desenho OO (*Object-Oriented*) [Brandon, 2002], tendo sido propostas algumas abordagens possíveis para adoptar os conceitos OO nas matrizes. De acordo com os princípios do OO, cada classe deveria encapsular o seu estado interno, recorrendo a métodos para permitir acções como *create, read*, *update* e *delete* a cada objecto. Para a linguagem Java e no contexto desta dissertação, cada uma destas acções foi definida do seguinte modo:

- Create – o constructor da classe foi chamado e é criado um objecto;
- Read – é chamado um selector num objecto;
- Update – é chamado um modificador num objecto;
- Delete – o objecto torna-se inacessível, ficando propenso a ser reciclado pelo *garbage collector*.

As matrizes CRUD OO podem ter vários níveis de granularidade. Normalmente, representam nas linhas as classes e nas colunas os casos de utilização, descritos nos diagramas de casos de utilização, ou os seus cenários. Existem outras abordagens que colocam nas linhas os métodos, em vez das classes [Brandon, 2002].

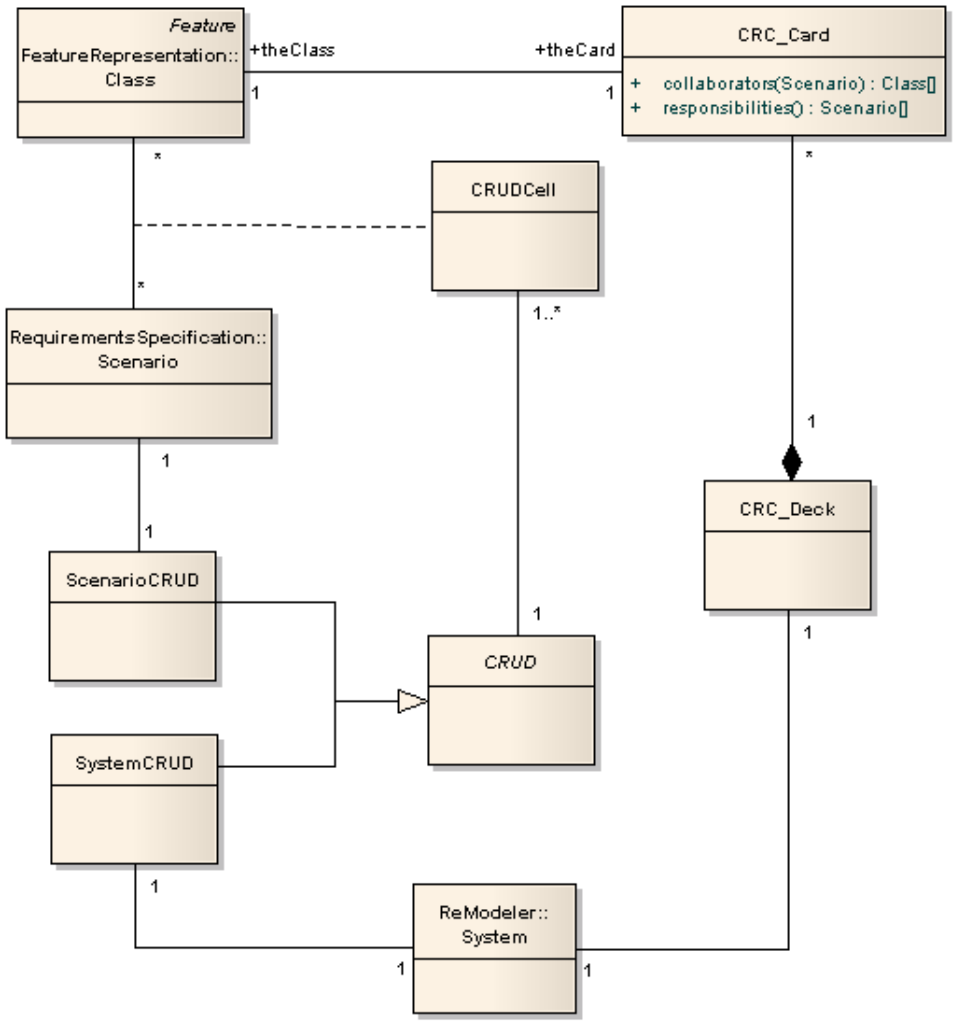


Figura 8. Características da Modelação Aumentada.

Nesta dissertação são propostos quatro tipos de matrizes CRUD que diferem entre si na granularidade e características apresentadas, de modo a melhor satisfazer as necessidades de quem as analisa. De um modo geral, cada matriz proposta representa um ou o conjunto de todos os cenários agrupados por casos de utilização (ver Figura 8). Cada célula da matriz vai referenciar um cenário e uma classe (salvo as variações a seguir descritas).

Na forma mais simples, a matriz apresenta apenas uma coluna, que corresponde a um cenário de execução, ou seja, a uma captura de um cenário, e apresenta apenas as classes que participaram realmente na captura (ver Figura 9).

Matriz CRUD do cenário *Change ball radius main scenario*

Classes / scenario	Change ball radius main scenario
MainPackage.PongFrame	U
MainPackage.Table	RU

Generated by ReModeler

Figura 9. Matriz CRUD para captura única.

Esta matriz mostra que no cenário "*Change ball radius main scenario*" estiveram envolvidas duas classes, *PongFrame* e *Table*, que pertencem ambos ao pacote *MainPackage*, e que foram executadas acções de leitura e de escrita. Para permitir uma visão global do sistema foram geradas matrizes que apresentam todos os cenários, agrupados pelos respectivos casos de utilização (ver Figura 10).

Estas matrizes representam as várias classes do sistema nas linhas (agrupadas por pacotes) e os cenários de utilização nas colunas (agrupados por casos de uso), identificando em cada célula quais os tipos de operações que foram executadas. Elas mostram todos os cenários definidos para todos os casos de utilização e respectivos resultados das capturas. Ao contrário da matriz anterior, nesta são mostradas todas as classes do sistema, independentemente de terem sido efectuadas acções nas mesmas.

Matriz CRUD do sistema

Use Cases	Play Game	Change game background color	Change ball radius	Modify name size		Change horizontal speed.
Classes /scenarios	Play game main scenario	Change game back color main scenario	Change ball radius main scenario	Modify name size main scenario	Modify name size alternative scenario	Change horizontal speed main scenario
MainPackage.Main						
MainPackage.PongFrame	U	U	U			U
MainPackage.PongMenu				CUD		
MainPackage.Table	CRU	U	RU	U	U	U

Generated by ReModeler

Figura 10. Matriz CRUD para sistema completo.

Apesar das matrizes apresentadas até agora terem granularidade ao nível da classe, existem situações em que é útil ter uma visão mais ou menos compacta do sistema. Torna-se portanto indispensável permitir a visualização das acções capturadas não só ao nível das classes, mas também dos pacotes ou dos métodos. Deste modo, a terceira matriz gerada permite ao utilizador escolher os elementos, e a respectiva granularidade, com que se quer analisar os resultados (ver Figura 11).

Matriz CRUD do sistema

Use Cases	Play Game	Change game background color	Change ball radius	Modify name size		Change horizontal speed.
Classes /scenarios	Play game main scenario	Change game back color main scenario	Change ball radius main scenario	Modify name size main scenario	Modify name size alternative scenario	Change horizontal speed main scenario
MainPackage	CRU	U	RU	CUD	U	U
MainPackage.Table	CRU	U	RU	U	U	U
MainPackage.Table.paintComponent()	U	U	U			

Generated by ReModeler

Figura 11. Matriz CRUD com diferentes níveis de granularidade.

Na matriz CRUD apresentada atrás é possível verificar que para as linhas, o nível de granularidade vai variando. Na primeira linha está um pacote, na segunda está um sub pacote, na terceira está uma classe e na última está um método.

Por último, é proposta uma outra matriz que adiciona informação extra à observada nas matrizes anteriores. Esta matriz mostra o número de vezes que cada acção ocorre, agrupado quer por linhas, quer por colunas (ver Figura 12).

Matriz CRUD do sistema TESTE

Use Cases	Play Game	Change ball radius	Modify name size		Totals
Classes /scenarios	Play game main scenario	Change ball radius main scenario	Modify name size main scenario	Modify name size alternative scenario	
MainPackage.Main					C(0) R(0) U(0) D(0)
MainPackage.PongFrame	U	U			C(0) R(0) U(2) D(0)
MainPackage.PongMenu			CUD		C(1) R(0) U(3) D(1)
MainPackage.Table	CRU	RU	U	U	C(1) R(2) U(10) D(0)
Totals	C(1) R(1) U(4) D(0)	C(0) R(1) U(5) D(0)	C(1) R(0) U(4) D(1)	C(0) R(0) U(2) D(0)	

Generated by ReModeler

Figura 12. Matriz CRUD com informação de quantidade.

Na matriz anterior é possível concluir, por exemplo, que a classe "*MainPackage.Table*" executou em todas as capturas um total de duas acções de leitura e dez acções de escrita. Por sua vez, para concretizar o cenário "*Play Game main scenario*" foram realizadas quatro acções de escrita e uma de leitura.

Em linguagens orientadas a objectos, como o Java por exemplo, existe ainda o problema da definição do que é um *delete*. Na verdade é o sistema de *garbage collector* que "apaga" os objectos e não uma acção efectuada directamente. Para colmatar este problema foi definido um âmbito para os objectos criados. Se um objecto é criado no âmbito de um método e não é retornado, pode-se dizer que no final do método o objecto é apagado porque não faz sentido fora do âmbito (ver

Figura 13). Quando isto acontece assume-se que ocorreu um *Delete* no diagrama de sequência respectivo e consequentemente uma acção de *Delete* na matriz de CRUD.

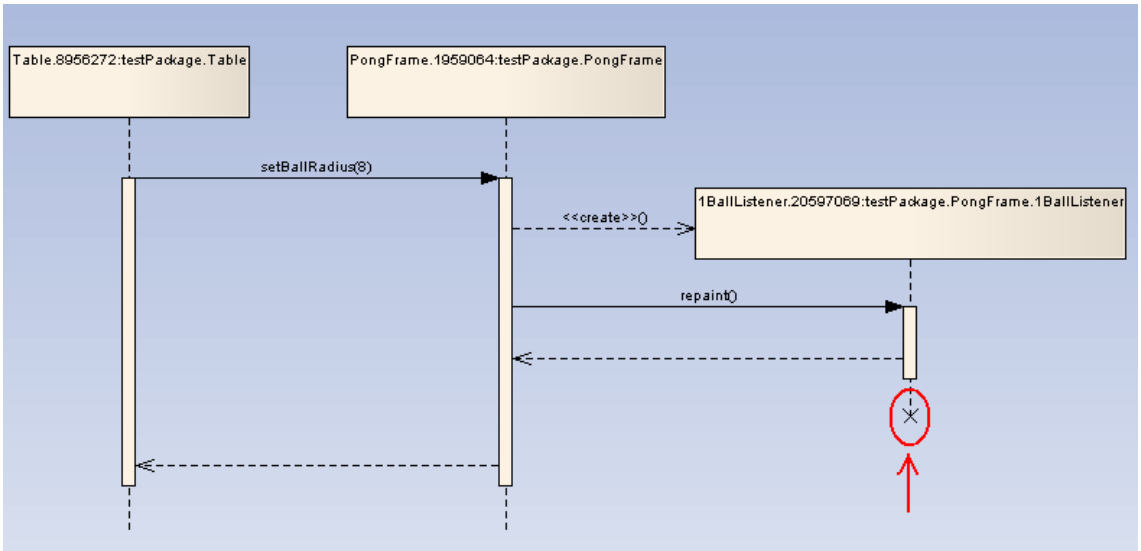


Figura 13. Diagrama de sequência representando o âmbito da criação de um objecto e respectivo *Delete*.

2.4 Cartões CRC Estendidos

Os cartões CRC (*Class-Responsabilities-Collaborations*) foram originalmente introduzidos em 1989 por Kent Beck e Ward Cunningham [Beck and Cunningham, 1989] como uma técnica de ensino do paradigma orientado a objectos, mas rapidamente foi estendida e proposta como uma técnica de elicitação de requisitos. Os cartões CRC passaram a ser uma aproximação informal à modelação orientada a objectos. São criados através dos cenários, baseados nos requisitos do sistema, que modelam o comportamento do mesmo. A técnica de modelação com os cartões permite identificar e organizar, de um modo simples, as classes que são relevantes para o sistema.

Um cartão CRC é usado para representar as responsabilidades de uma classe e as interacções entre classes. Um modelo CRC é uma colecção destes cartões. Os cartões estão divididos em três secções (ver Figura 14). No topo do cartão escreve-se o nome da classe, no corpo do cartão são listadas as responsabilidades à esquerda e as colaborações à direita. Cada responsabilidade representa um serviço que a classe deve disponibilizar, algo que a classe conheça ou saiba. Os colaboradores são outras

classes do sistema que colaboram com a classe em questão para garantir a prossecução de uma dada responsabilidade.

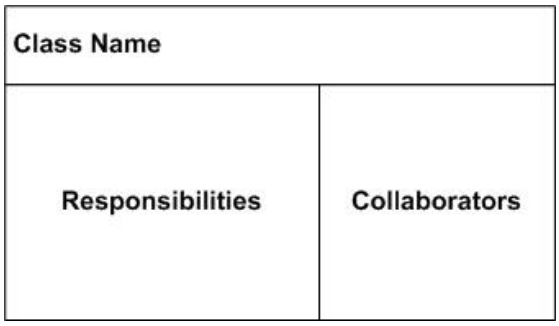


Figura 14. Layout de um cartão CRC.

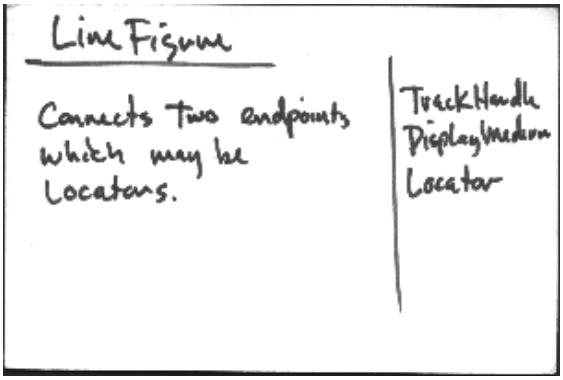
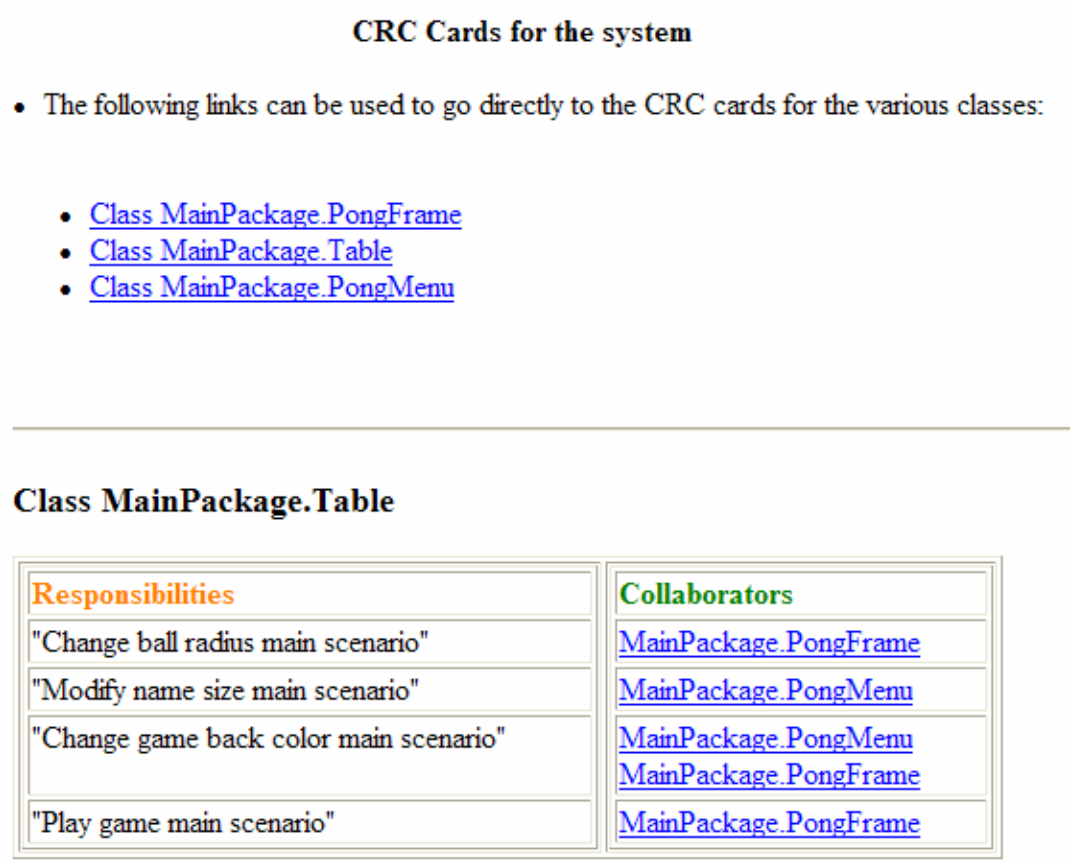


Figura 15. Cartão CRC original (manual).

Nos cartões CRC propostos, as responsabilidades representam os cenários executados e as colaborações visam a concretização dos mesmos. Estes cartões apresentam algumas diferenças dos originalmente concebidos, não só no modo e contexto em que são criados, mas no modo em como são apresentados. Originalmente os cartões eram cartões físicos (manuais) e tinham como principais preocupações serem portáveis e permitirem aos intervenientes ter a percepção de como o sistema se iria comportar (ver Figura 15). Os cartões propostos têm as mesmas preocupações que os anteriores, mas de um modo diferente (ver Figura 16). Os cartões são em formato HTML, o que permite a sua portabilidade para qualquer sistema e a navegação entre as classes colaboradoras e os respectivos cartões. Estes cartões permitem auxiliar na manutenção do software, mostrando de uma forma simples como os blocos arquitecturais (classes) colaboram entre si para implementar cada requisito de negócio. A visão rápida que proporcionam, permite identificar que impacto poderá ter uma alteração numa determinada classes ou num requisito e

permite identificar *bottlenecks* do sistema, bem como classes que estão a ser usadas de um modo muito sobrecarregado e por isso merecem ser optimizadas. Ao observar a Figura 8, é possível perceber que cada cartão proposto representa uma classe e que o seu conjunto forma uma pilha que representa todo o sistema.

Por estas razões, os cartões CRC apresentados nesta dissertação foram denominados de *Extented CRC Cards*.



CRC Cards for the system

- The following links can be used to go directly to the CRC cards for the various classes:

  - Class MainPackage.PongFrame
  - Class MainPackage.Table
  - Class MainPackage.PongMenu

Class MainPackage.Table

Responsibilities	Collaborators
"Change ball radius main scenario"	MainPackage.PongFrame
"Modify name size main scenario"	MainPackage.PongMenu
"Change game back color main scenario"	MainPackage.PongMenu MainPackage.PongFrame
"Play game main scenario"	MainPackage.PongFrame

Figura 16. Extended CRC Card.

2.5 Diagramas UML coloridos

O aumento da competitividade tem forçado as empresas a diminuir os prazos de entrega dos seus produtos, ao mesmo tempo que devem aumentar a qualidade e a redução dos seus custos. Garantir que os seus produtos funcionam de acordo com os requisitos é indispensável para o aumento da qualidade. Torna-se necessário testar um sistema para se conseguir produzir software fiável e obedecendo aos requisitos [Delgado, 2006].

As principais medidas de um teste incluem a cobertura e a qualidade do próprio teste. A cobertura é a medida da abrangência do teste e é expressa pela cobertura dos requisitos e casos de teste ou pela cobertura do código executado. A qualidade é uma medida de confiança, de estabilidade e de desempenho do objectivo do teste, baseando-se na avaliação dos resultados do teste e na análise dos defeitos identificados [Rational, 2001]. Por esta razão, várias ferramentas têm sido desenvolvidas ao longo dos últimos anos, na tentativa de suportar, não só o processo de testes, mas também a identificação das partes de um sistema que não estão a ser testadas.

Nesta dissertação são propostos dois tipos de testes de cobertura: um é baseado nos requisitos e o outro baseado no código. O primeiro tipo funciona para um conjunto de casos de utilização e respectivos cenários - *Scenario Test Battery* [Gouveia, 2008], em que interessa ao analista identificar quais os cenários que já foram analisados, ou seja, capturados no sistema. O segundo tipo relaciona-se com as partes do sistema de foram realmente executadas, e com que intensidade, assemelhando-se à abrangência de execução do código fonte.

A representação dos resultados de testes de cobertura é muito variável, podendo ir desde o cálculo de uma valor percentual ou numérico, a representações gráficas ou esquemáticas. No entanto, algumas destas representações não são fáceis de analisar ou compreender.

Uma vez que os diagramas da UML disponibilizam várias visões sobre um sistema e que alguns deles constituem uma base de funcionamento do processo descrito, faz sentido que os resultados dos testes de cobertura sejam representados também nestes mesmos diagramas. A ideia é adicionar uma metáfora de cores nos diagramas tradicionais, de modo a representar os resultados relativos à cobertura.

Para o primeiro tipo descrito anteriormente, a representação é feita através de uma palete de cores, cuja intensidade demonstra a percentagem de cobertura. No exemplo da Figura 17 é possível verificar que apenas os casos de utilização que estão preenchidos a verde estão totalmente cobertos, ou seja, já foram capturados todos os seus cenários.

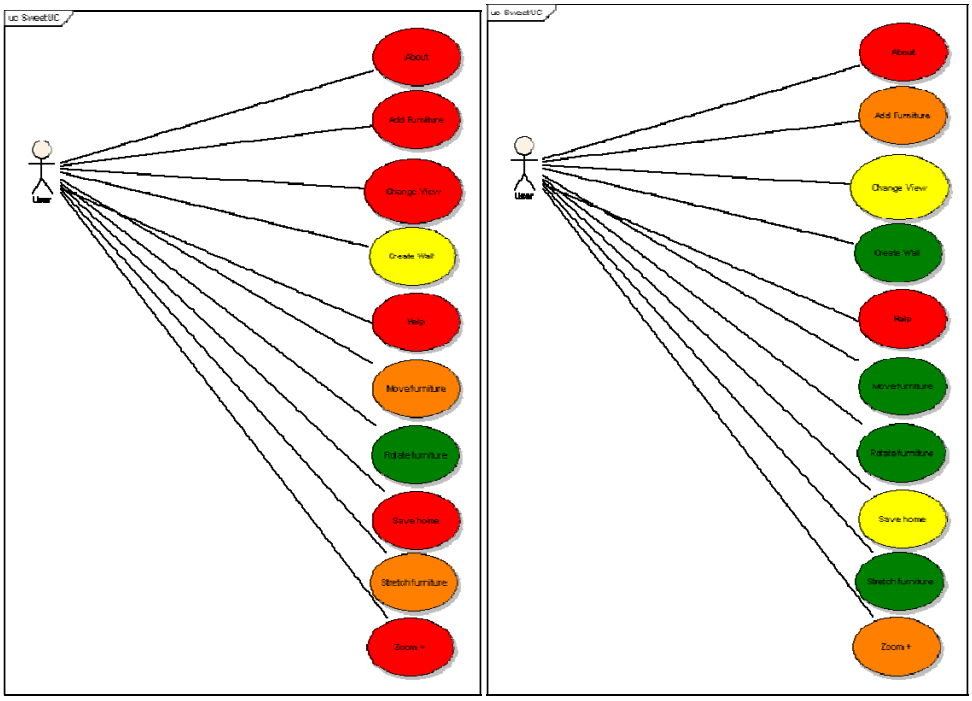


Figura 17. Colored Use Case Diagrams.

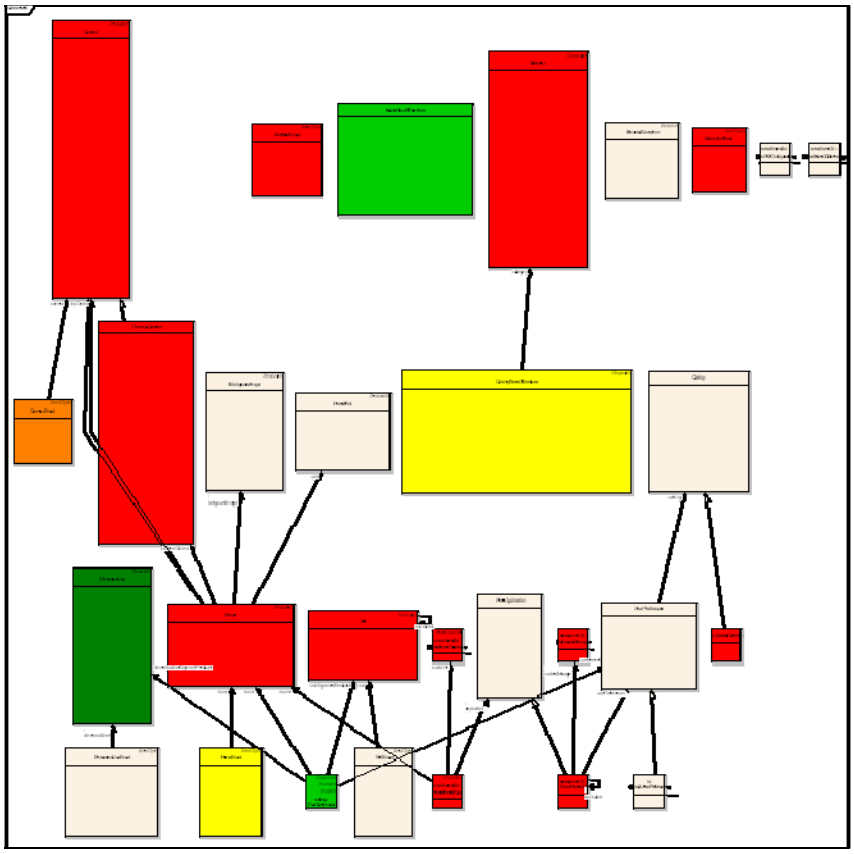


Figura 18. Colored Class Diagram.

Para o segundo tipo, a representação é bastante idêntica. É definida uma palete de cores que devem exprimir a execução das classes, tornando-se mais ou menos intensas consoante a sua utilização. O resultado é exportado em formato XMI, segundo diagramas de classes coloridos. Estes diagramas podem indicar a cobertura de execução da classe, analisando os métodos que são executados em comparação com os que não são, mas também a intensidade de utilização de cada classe, para identificar *bottlenecks* e analisar os módulos que permitem o melhoramento do desempenho. Um destes diagramas, denominado de *Colored Class Diagrams*, é mostrado na Figura 18.

Recorrendo novamente à ontologia do domínio, é possível esquematizar os diagramas propostos como mostra a Figura 19. Na verdade, qualquer um dos diagramas propostos é uma extensão a diagramas UML que representam classes ou casos de utilização, respectivamente (ver Figura 20).

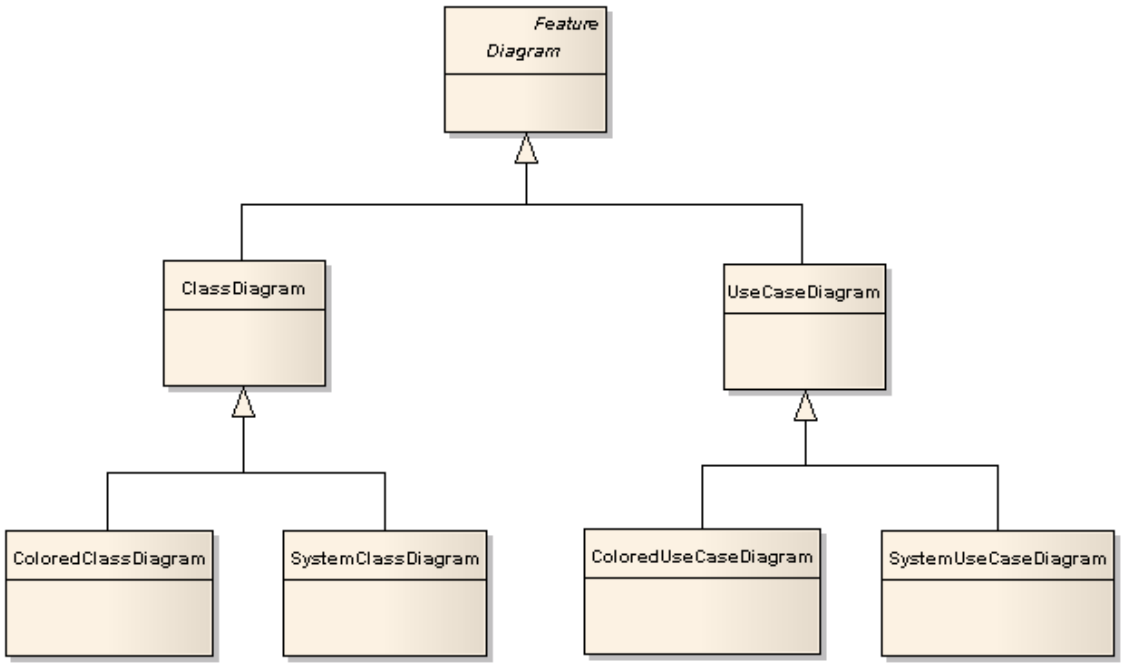


Figura 19. *Diagrams*.

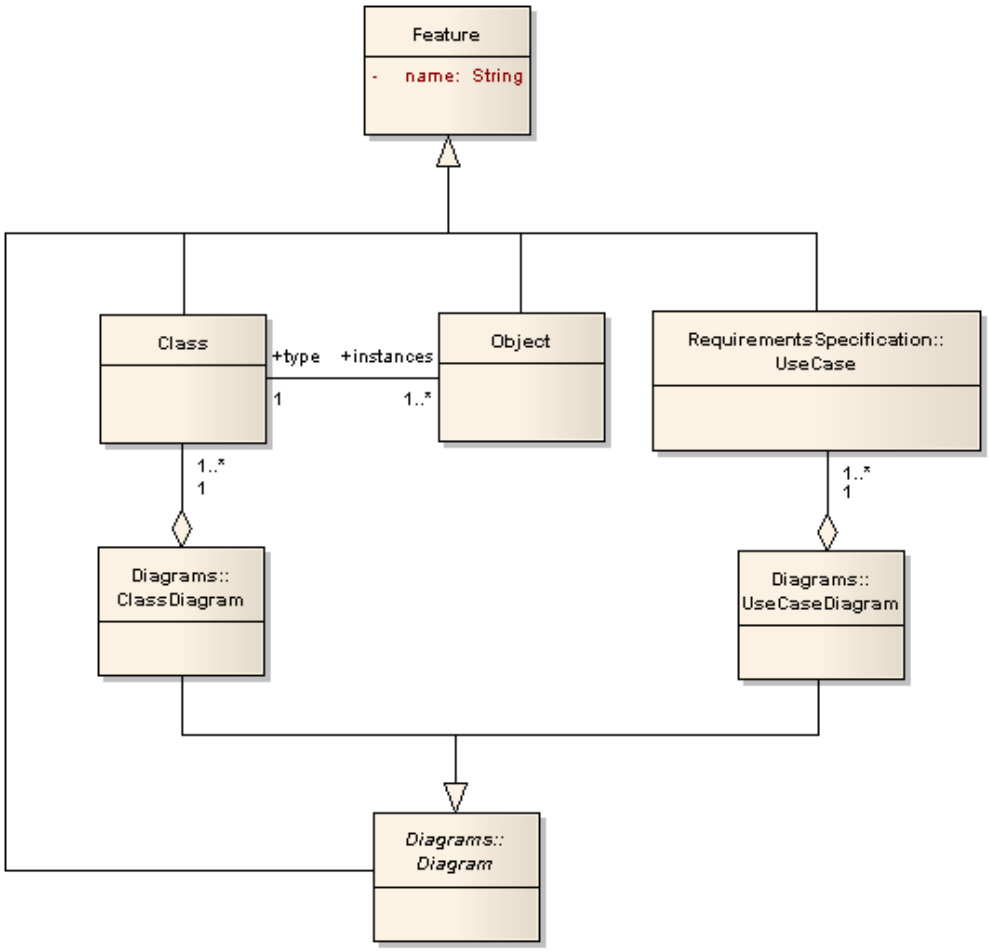


Figura 20. Representação dos componentes base.

2.6 Processo do *ReModeler*

Os artefactos, referidos na secção 1.4 deste capítulo, são gerados no âmbito do processo do *ReModeler* proposto nesta dissertação. O processo do *ReModeler* pode ser decomposto em vários sub-processos que se interligam, como mostra o diagrama da Figura 21. Este diagrama permite perceber as dependências entre as várias actividades que constituem o processo.

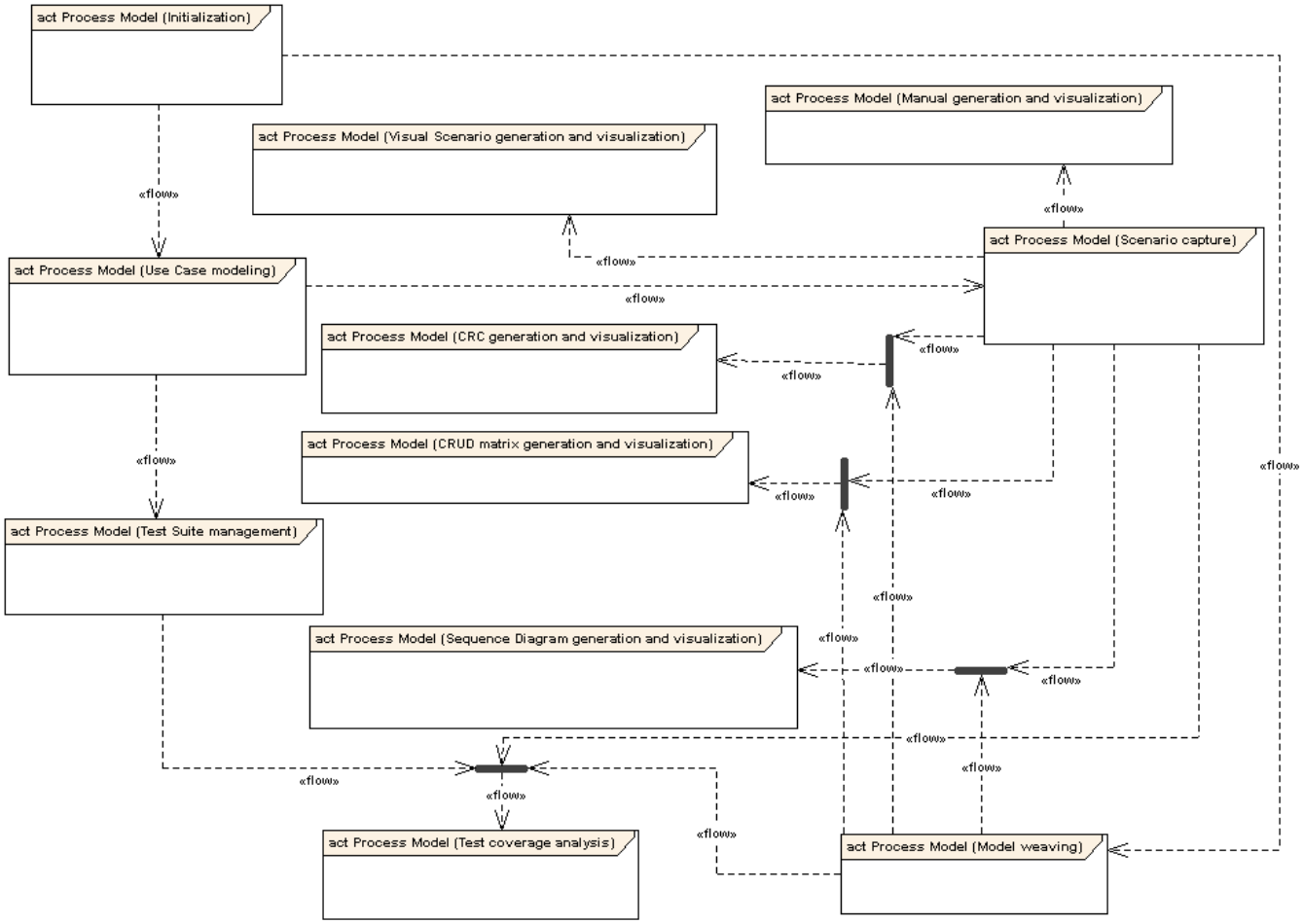


Figura 21. Modelo de processo global.

O início do processo do *ReModeler* é representado pelo processo *Initialization* (ver Figura 22). Este processo vai inicializar o sistema a usar, realizando actividades como a escolha do sistema a analisar e o seu entrelaçamento com o sistema *ReModeler*. Nenhuma funcionalidade pode ser levada a cabo, sem este passo inicial. Depois de concluídas as actividades descritas no processo anterior, outros dois processos estão prontos para serem executados: o *Model Weaving* e o *Use Case Modeling*.

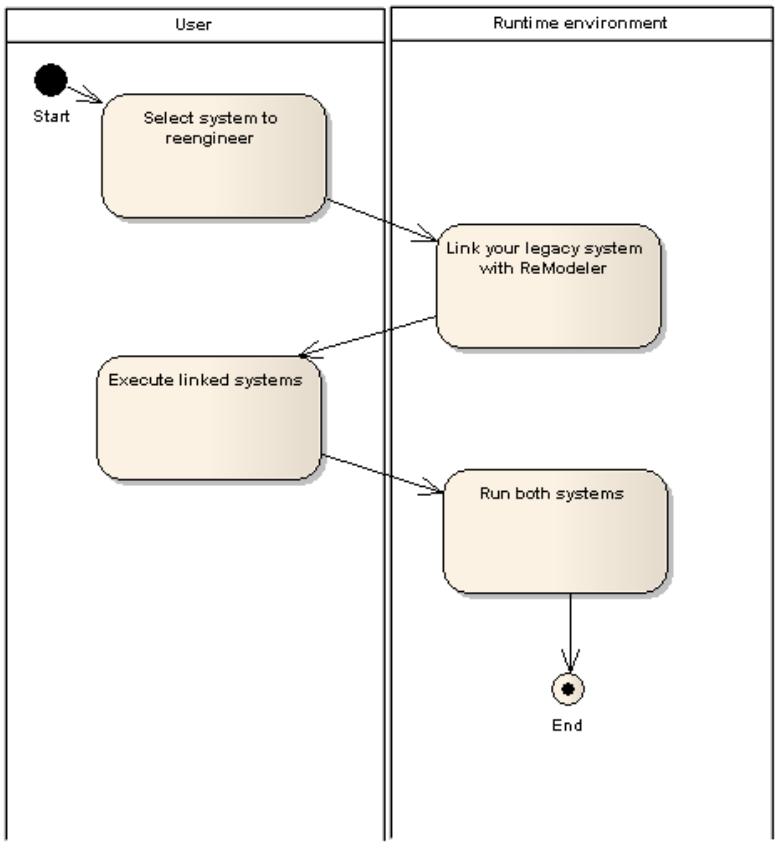


Figura 22. Processo de Inicialização.

O processo *Model Weaving* vai, a partir das facilidades de *reverse engineering* de uma ferramenta externa, recuperar o modelo de classes estático do sistema em análise (ver Figura 23). O modelo é depois exportado em formato XMI para permitir a sua importação no *ReModeler*. Este modelo é de extrema importância para a concretização de vários outros processos, como mostra a Figura 21.

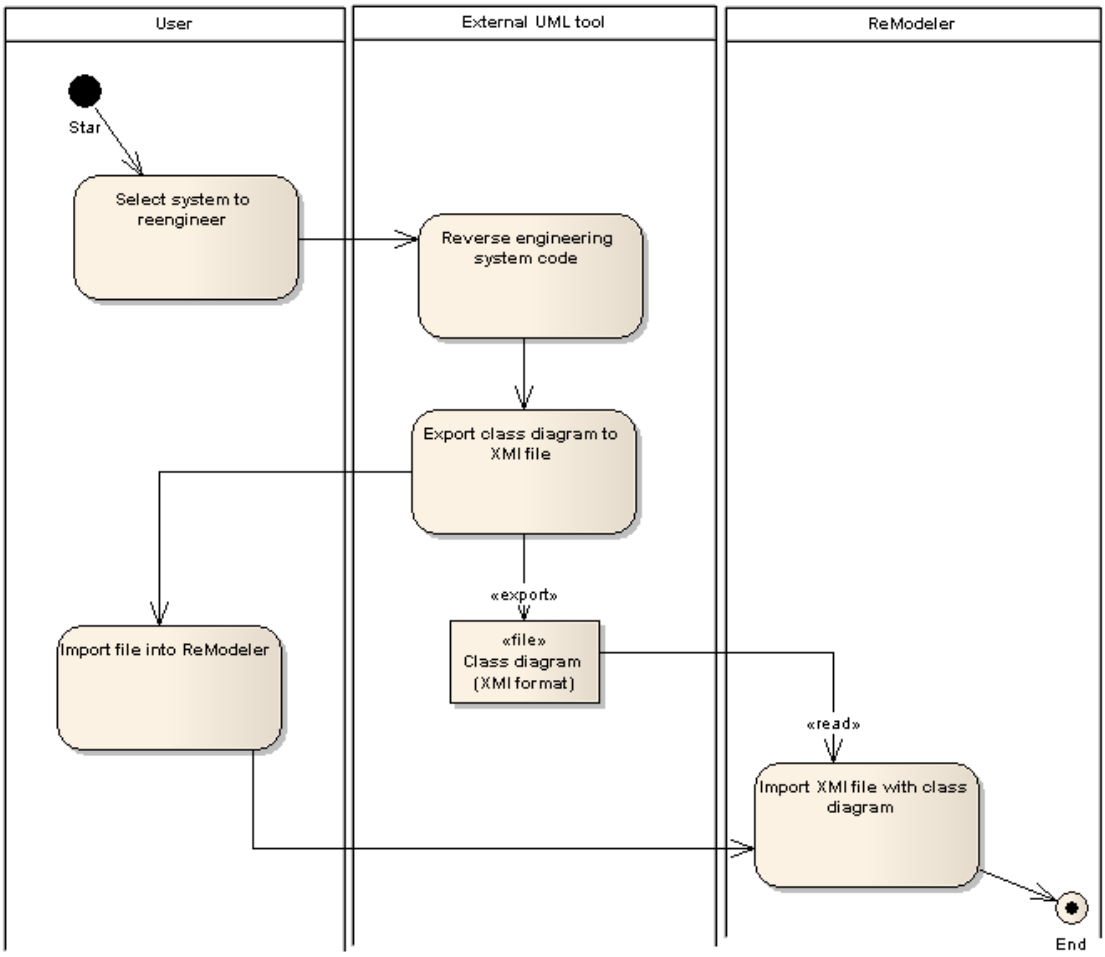


Figura 23. Processo de recuperação e importação de modelos.

O processo *Use Case Modeling* descreve as actividades que devem ser levadas a cabo para documentar os requisitos funcionais do sistema, como mostra a figura 23. Existem duas maneiras de introduzir essa informação no sistema: (i) através da construção directa dos casos de utilização na ferramenta *ReModeler* ou (ii) através da importação de um diagrama de casos de utilização, em formato XMI, criado numa ferramenta UML externa. Para qualquer dos casos, é necessário criar e descrever, através de um conjunto de passos, os cenários que correspondem aos casos de utilização.

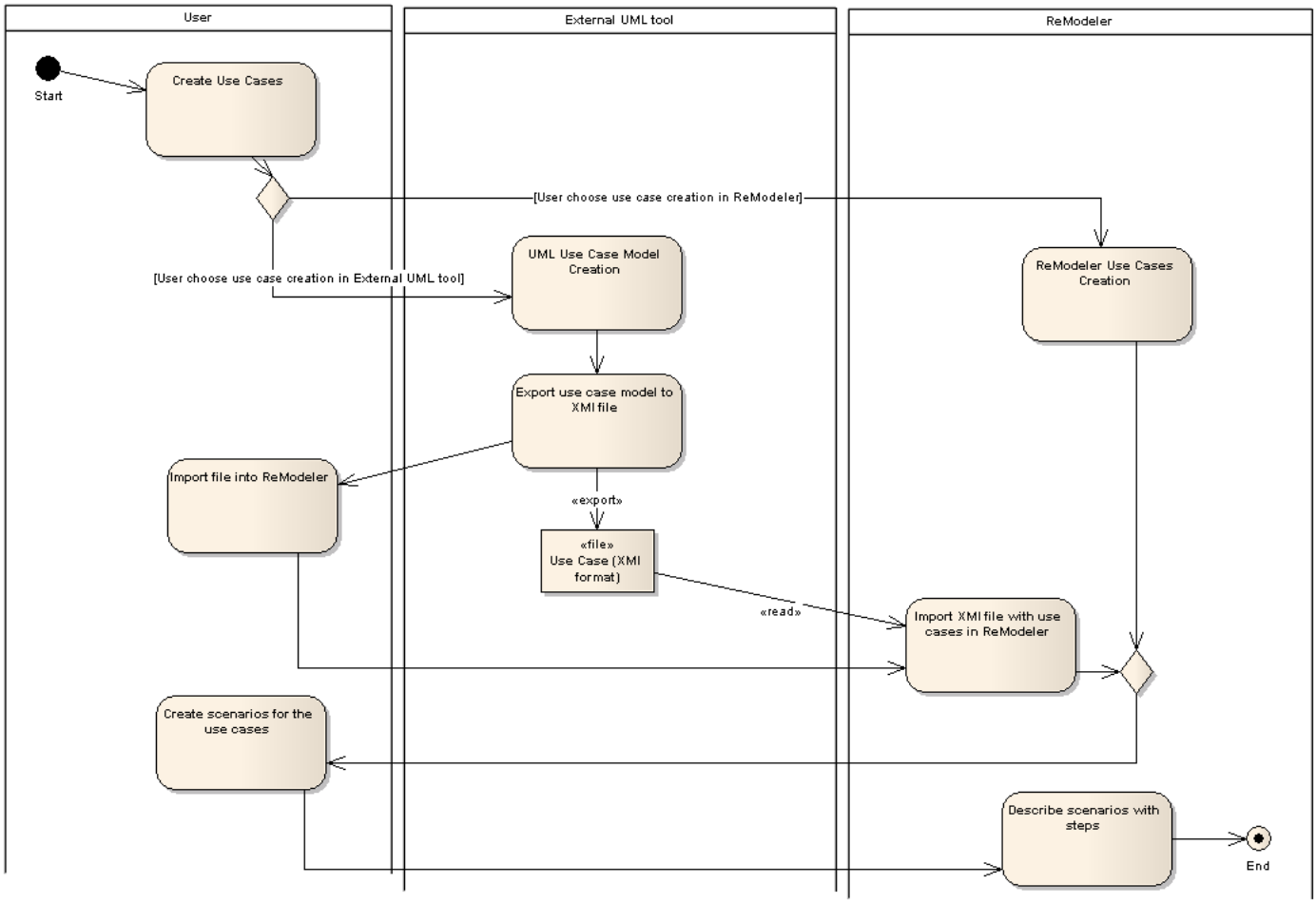


Figura 24. Processo de Modelação de casos de utilização.

Um dos processos que pode seguir a execução do processo do *ReModeler* é o *ScenarioCapturer*, mostrado na figura 24. O processo *ScenarioCapturer* agrega todas as actividades necessárias para proceder à captura de cada cenário, separadamente, que constitui a base de todos os artefactos propostos, como mostra a Figura 21. A captura de um cenário é composta pela captura interna e pela captura externa, sendo que a captura externa refere-se à produção do filme das interacções do utilizador e é descrito em [Gouveia, 2008].

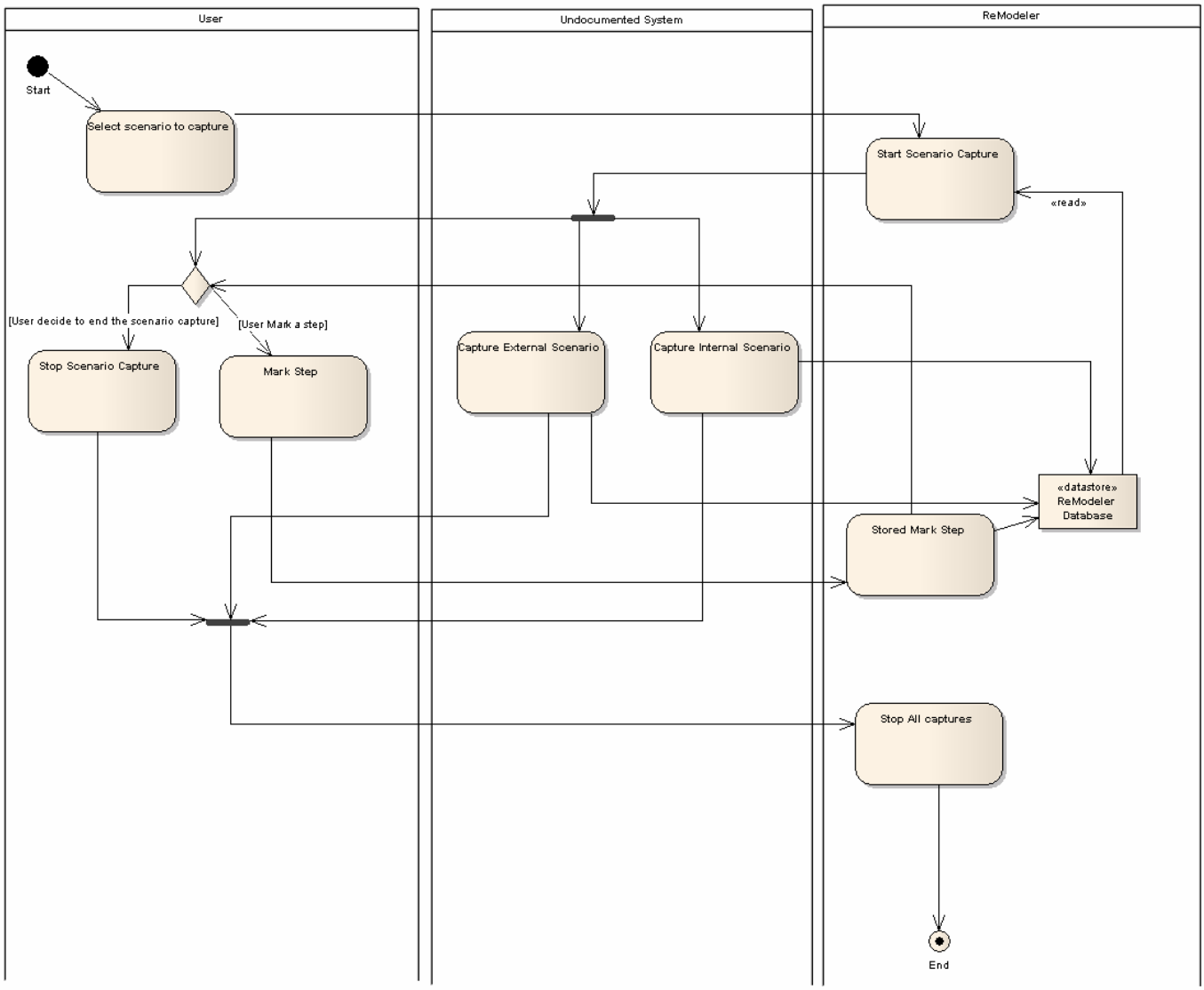


Figura 25. Processo de captura de cenários.

A captura interna de cenários pode ocorrer em dois modos distintos. Ou se está a desenvolver um novo sistema, entregando incrementalmente casos de utilização, ou já se tem um sistema (legado). No primeiro caso, podem-se capturar os cenários de um novo caso de utilização sempre que este se considerar estável. No segundo caso podem-se capturar todos os cenários numa única sessão. A execução de um cenário traduz-se internamente na troca de mensagens entre os objectos do sistema. Tal como representado na Figura 26, a captura de um cenário envolve um actor: o perito no domínio (analista do negócio ou utilizador final). O perito no domínio executa cada cenário separadamente. Ele vai executar o sistema através do cenário escolhido, passo por passo, como se estivesse a fazer uma utilização normal

do sistema, enquanto que numa janela separada ele marca qual dos passos do cenário ele está a executar em cada momento.

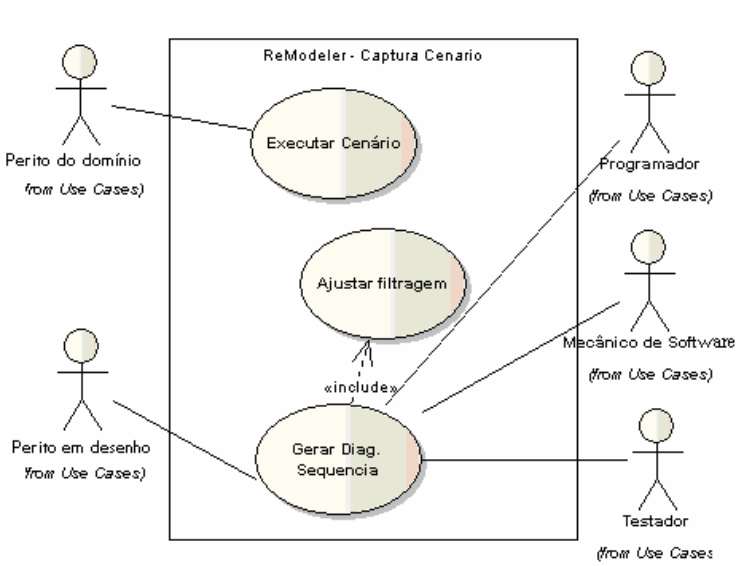


Figura 26. Casos de Utilização para a Captura Interna.

Os dados recolhidos em cada captura são armazenados directamente na base de dados do *ReModeler*, para posteriormente permitirem gerar todos os artefactos já focados. Um desses artefactos é o diagrama de sequência temporizado, que é gerado pelo processo *Sequence Diagram generation and visualization* (ver Figura 27). Em cada captura, o volume de informação que se extrai pode ser bastante grande, mesmo para sistemas relativamente pequenos. Em apenas alguns minutos, é possível recolher milhares de objectos e milhões de invocações de métodos. Isto torna a representação e análise dessa informação praticamente impraticável, mesmo usando diagramas. Existem vários mecanismos para reduzir o volume da informação a representar. Neste processo é usado um mecanismo de filtragem de eventos. Através da possibilidade de seleccionar os eventos que se quer ver representados, o analista apenas vê a informação que considera relevante. Por esta razão, o processo de geração de diagramas de sequência temporizados inicia-se com as opções de filtragem, podendo o utilizador escolher entre criar uma nova filtragem (com a informação da captura ou do sistema na sua globalidade), ou utilizar uma já existente. Os diagramas gerados são depois exportados sob a forma de ficheiros

XMI, para permitir a interoperabilidade entre ferramentas, na medida em que este constitui um formato padrão para a representação de diagramas UML.

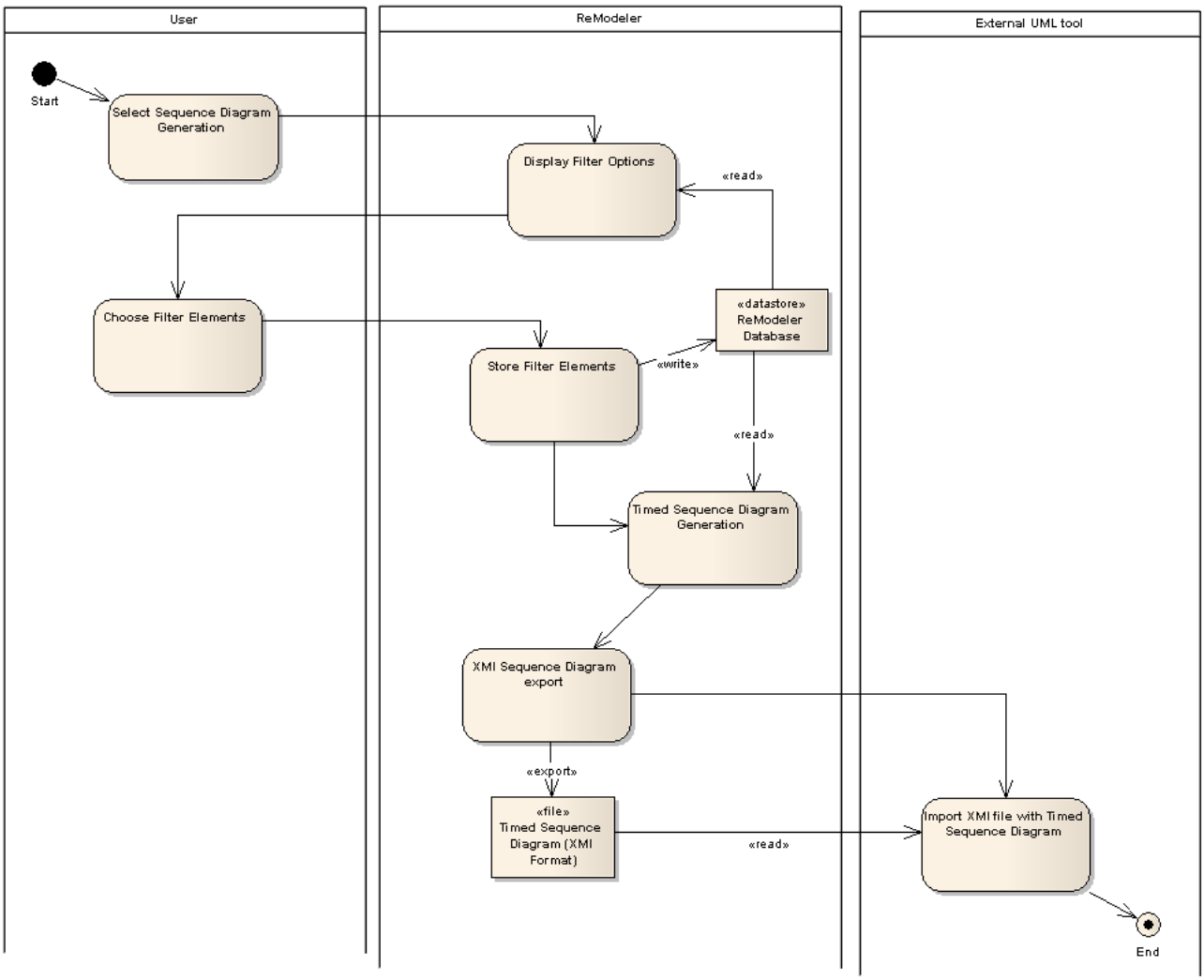


Figura 27. Processo de geração e visualização de diagramas de sequência.

Na Figura 26, podemos ver a parte do diagrama de casos de utilização do *ReModeler*, que representa a captura interna e respectiva geração de diagramas.

A captura de um cenário vai originar também a geração de uma matriz CRUD simples. A geração desta matriz, e das restantes propostas, é descrita no processo *CRUD matrix generation and visualization* (ver Figura 28).

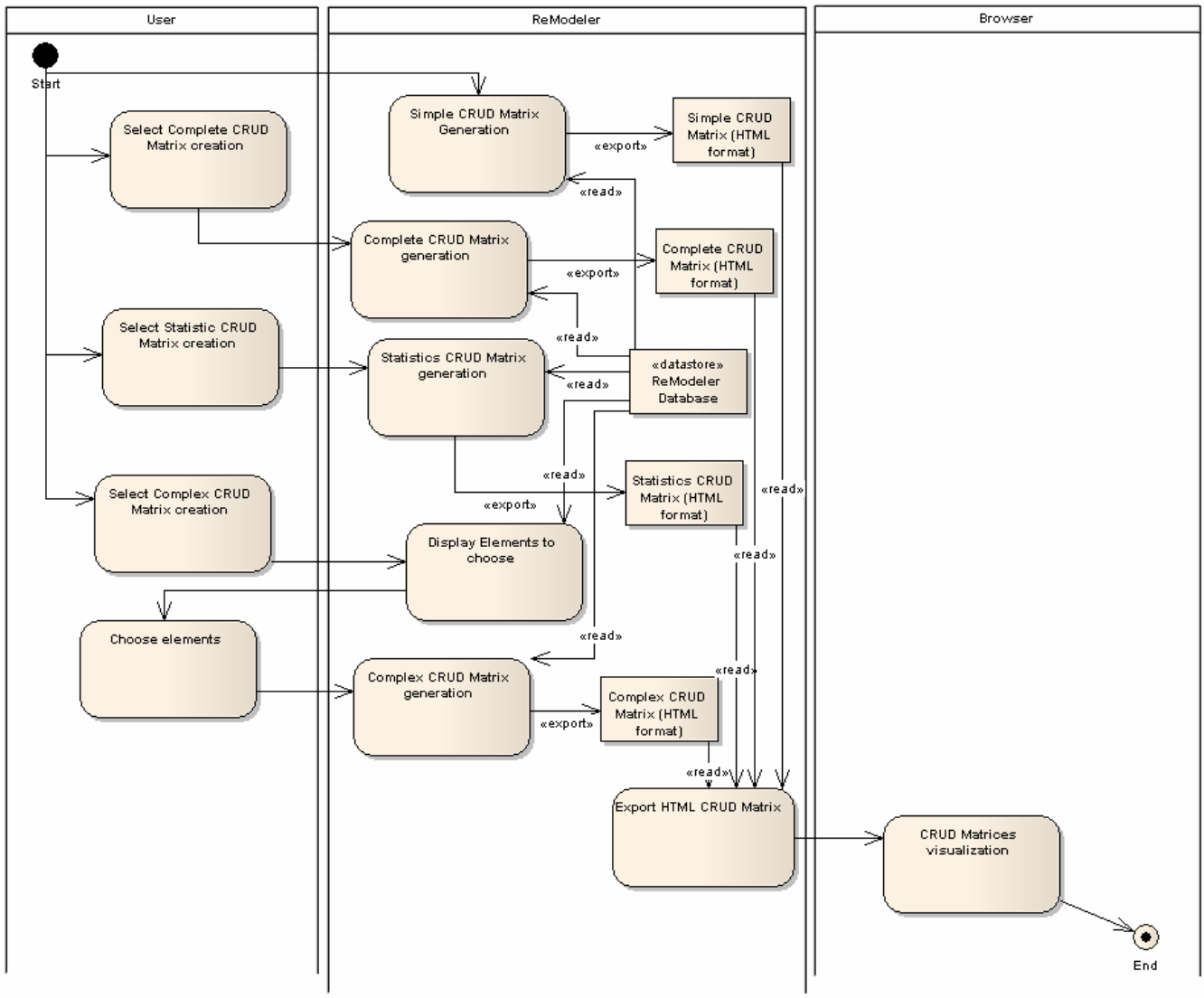


Figura 28. Processo de geração e visualização de Matrizes CRUD.

Usualmente uma matriz CRUD era criada manualmente, mas para sistemas grandes e complexos isso pode ser uma tarefa árdua e até desmotivante. É evidente que um mecanismo de geração automática de tal matriz é útil neste tipo de situações. Porém, tal só é possível na presença de um sistema construído, isto é numa fase mais tardia do ciclo de vida do software. A Figura 29 mostra os principais actores que interagem com a geração das matrizes.

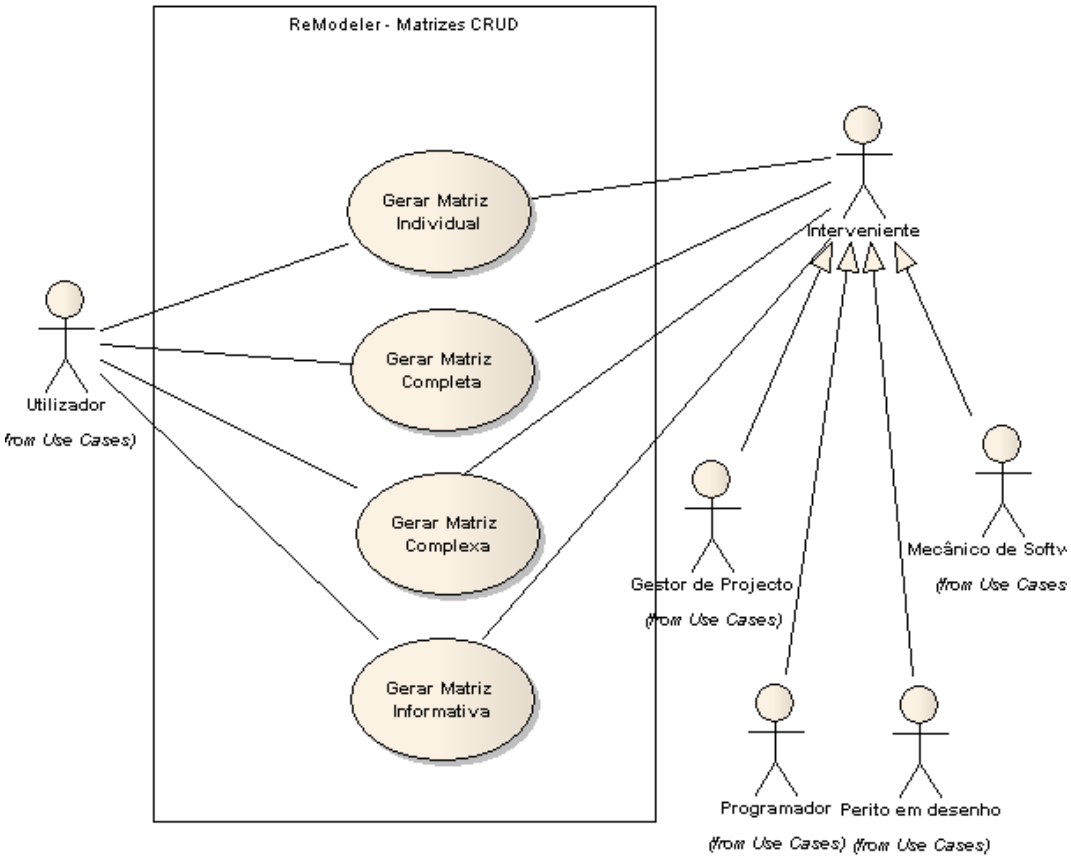


Figura 29. Casos de Utilização para a geração de matrizes CRUD.

No processo proposto esta geração já é possível, na medida em que existe normalmente o sistema, ou parte dele, construído e em funcionamento, que é capturado recolhendo as informações necessárias.

A geração das matrizes, com excepção da matriz CRUD simples, necessita da importação do diagrama de classes global do sistema, porque nas suas linhas vão aparecer todas as classes do sistema, independentemente de terem sido capturadas ou não, para tornar o resultado gerado mais realista e abrangente. A matriz CRUD complexa necessita ainda que o utilizador escolha os elementos: pacotes, classes ou métodos, que pretende ver representados na matriz.

Para além das matrizes CRUD, também os cartões CRC são gerados automaticamente, pelo processo *CRC generation and visualization* (ver Figura 30).

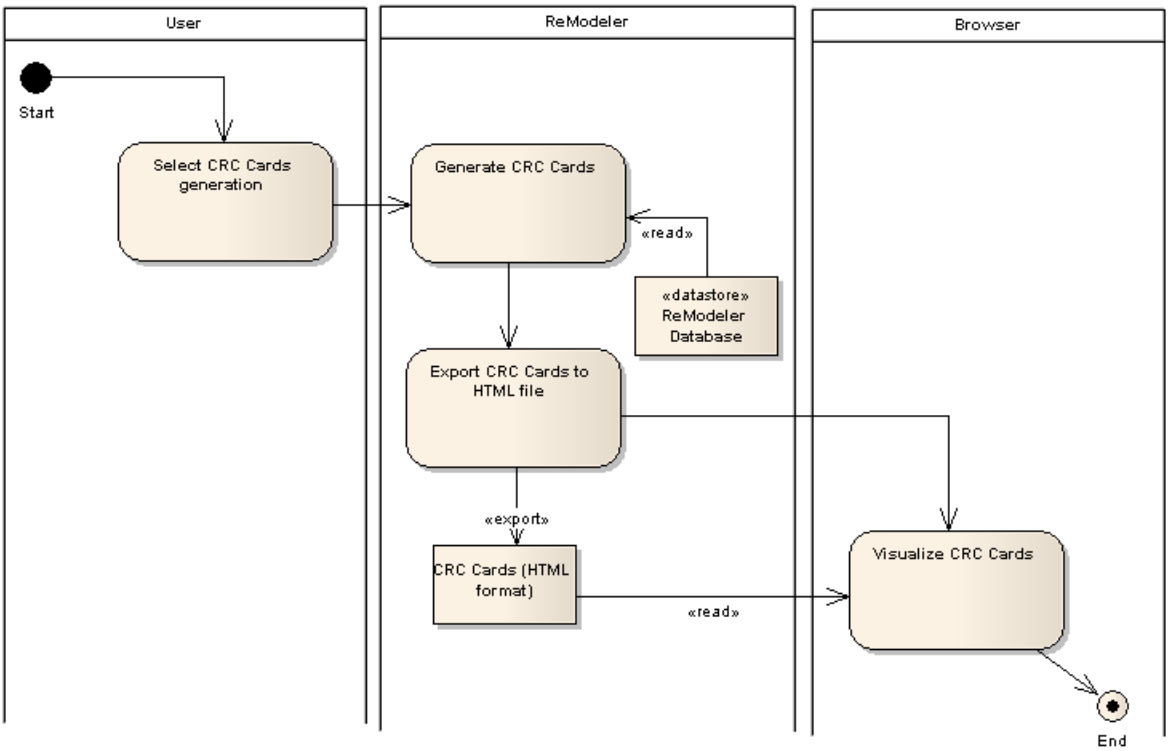


Figura 30. Processo de geração e visualização de cartões CRC.

Inicialmente estes cartões foram propostos como cartões manuais, cuja criação e modelação era feita em sessões. Nessas sessões estavam presentes vários tipos de pessoas envolvidas num projecto [Ambler, 1998], que interagiam numa espécie de encenação do que o sistema deveria fazer e do que cada classe deveria realizar para a concretização dos vários cenários. No entanto, para a compreensão e documentação de sistemas legados, esta técnica manual não é exequível, dado que seria necessário analisar o código fonte e seguir o seu traço de execução para reconstruir os cartões. No processo proposto, tal como para as matrizes CRUD, também os cartões CRC são gerados automaticamente em formato HTML de modo a permitir a navegação entre eles. Na Figura 31 é possível ver um excerto do diagrama de casos de utilização do *ReModeler* que mostra os intervenientes na criação destes cartões.

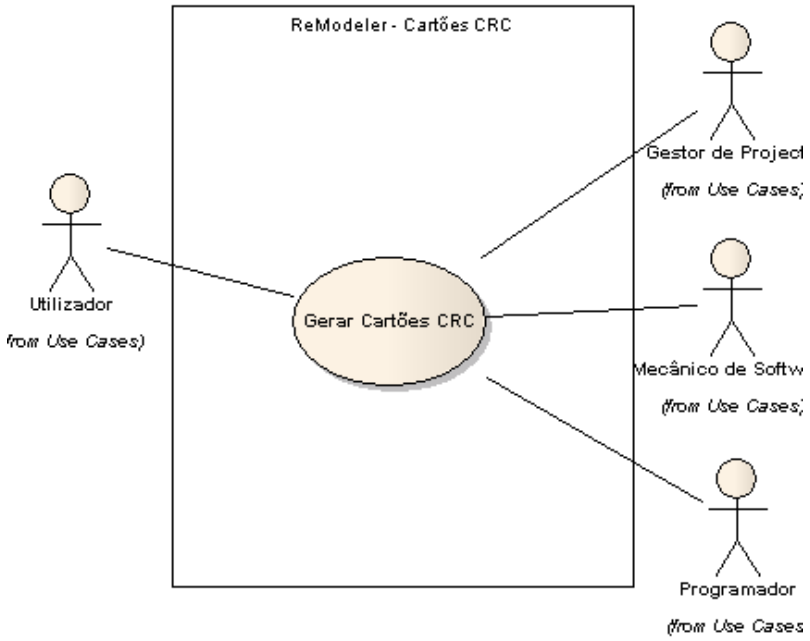


Figura 31. Casos de Utilização para a geração de cartões CRC.

Para que os cartões CRC representem todo o sistema a analisar, antes da sua geração, é necessário importar no *ReModeler* o diagrama de classes global (ver Figura 21).

A última funcionalidade disponibilizada pelo processo do *ReModeler*, no âmbito desta dissertação, é a análise de testes cobertura e intensidade de execução, detalhada no processo *Test coverage analysis* (ver Figura 32).

O processo *Test coverage analysis* apresenta dois tipos de teste de cobertura e intensidade: a cobertura das capturas dos cenários dos casos de utilização e a cobertura e intensidade de execução das classes do sistema.

Durante a tarefa de documentação do sistema, os Gestores de projectos e os Engenheiros de Requisitos vão querer saber qual tem sido o progresso da descrição da totalidade dos cenários para todos os casos de utilização. Eles também vão querer saber o progresso das capturas dos cenários. Estas necessidades são particularmente importantes para gestores de projectos, para terem algum tipo de controlo do projecto assim como para os engenheiros de requisitos quando trabalham em grandes projectos, onde provavelmente uma equipa, em vez de uma pessoa só, está a cargo da documentação dos requisitos. Outra ocasião em que esta funcionalidade traz grandes vantagens é quando existe alteração de equipas. Para mitigar estes problemas, são produzidos diagramas de casos de utilização UML onde está presente uma escala de cores que representa a percentagem de capturas de cenários para cada caso de

utilização. Para além disto, os programadores e os engenheiros de testes vão querer identificar que partes do sistema estão realmente a ser usadas e de que modo no contexto de um determinado cenário. Quando o sistema é muito grande ou muito complexo, eles podem querer ver este tipo de análise de cobertura ao nível de granularidade da classe para permitir um bom nível de abstracção. Para fazer face a este problema, são gerados diagramas de classe UML, em que cada classe tem uma cor de fundo de acordo com uma escala de cor de níveis de percentagem dos métodos, dentro da mesma, que tenham sido usados para um determinado cenário. Na Figura 33 podemos observar os intervenientes na geração dos diagramas, representados no excerto do diagrama dos casos de utilização do *ReModeler*.

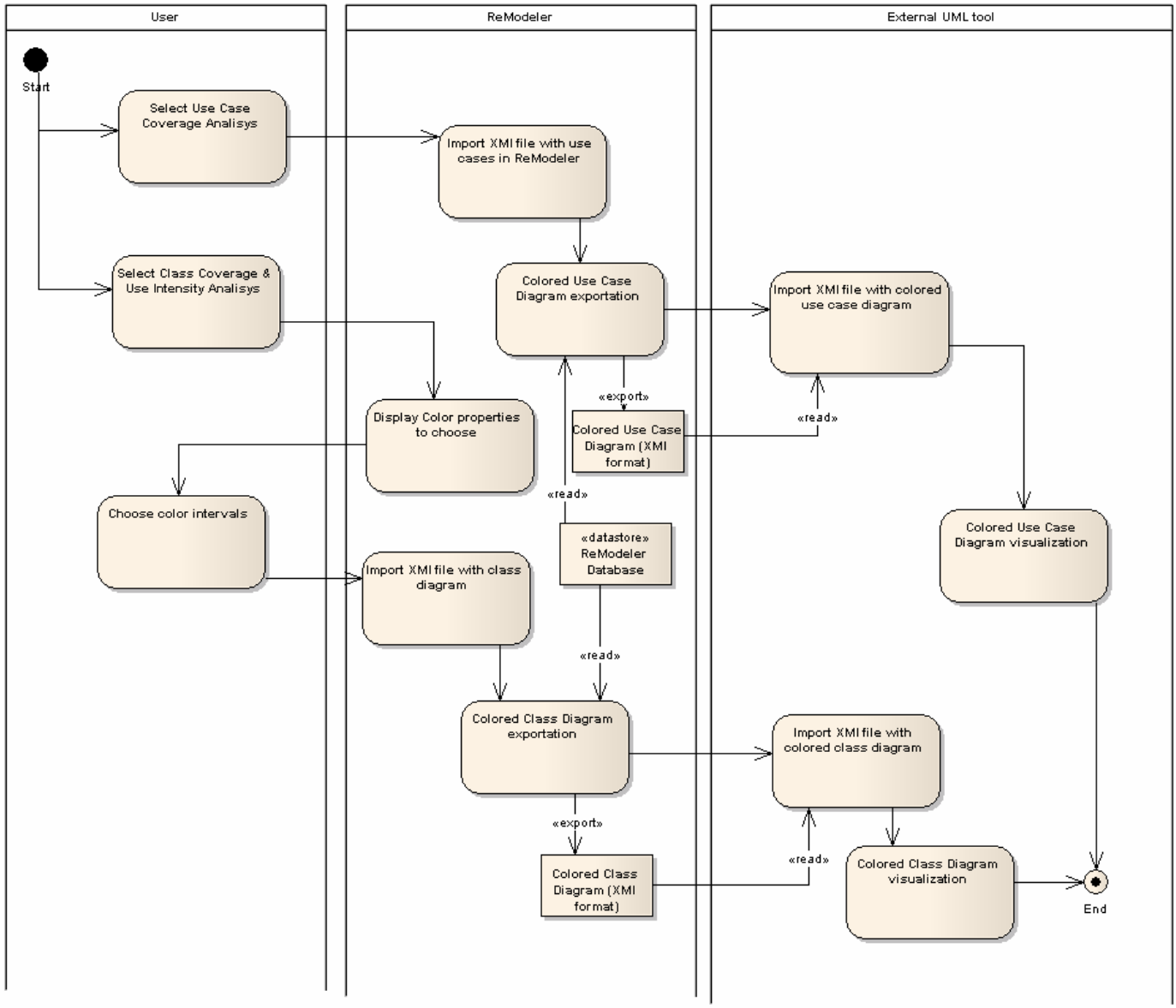


Figura 32. Processo de análise de testes de cobertura.

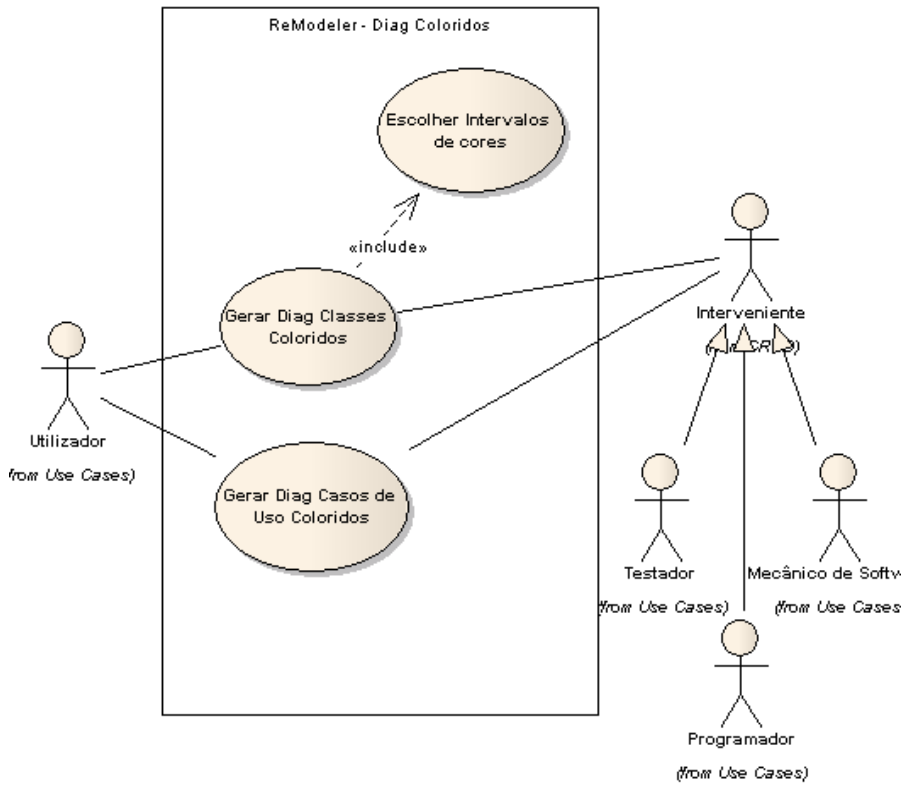


Figura 33. Casos de Utilização para a geração de diagramas coloridos.

As gerações dos diagramas coloridos são conseguidas pela análise da estrutura do sistema (capturada de forma estática por uma ferramenta especializada, das quais existem várias disponíveis no mercado) em comparação com a informação recolhida de forma dinâmica. Em qualquer dos casos, torna-se necessária a importação do diagrama original. Assim, para a geração de diagramas de casos de utilização coloridos é necessário importar o diagrama de casos de utilização original, em formato XMI, e o mesmo para o diagrama de classes. No caso da geração do diagrama de classes colorido é ainda dado a escolher ao utilizador uma palete de cores que devem exprimir a execução das classes, tornando-se mais ou menos intensas consoante a sua utilização. O resultado das gerações é exportado em formato XMI.

2.7 *Rational Unified Process* (RUP)

O processo proposto nesta dissertação, juntamente com os artefactos gerados, poderia ser integrado num *framework* de suporte ao desenvolvimento de software existente. A ênfase nos modelos UML e a definição de práticas para a produção de

software, suportadas por ferramentas, que automatizam partes do processo, fazem do *Rational Unified Process* (RUP) [Rational, 2001] um dos *frameworks* possíveis para essa integração.

O RUP define um processo iterativo de desenvolvimento de software que pode ser adaptado a cada organização. Este surgiu da análise das razões que levavam às falhas no desenvolvimento de software. O RUP propõe um conjunto de práticas que visam assegurar a produção de software de qualidade, que atinja as necessidades dos utilizadores finais, dentro dos prazos e orçamentos previstos. As actividades previstas no RUP, em vez de se focarem na produção de vários documentos, dão ênfase ao desenvolvimento e manutenção de modelos semanticamente ricos. Pode dizer-se que representa um guia de como usar eficientemente a UML. O RUP é ainda suportado por ferramentas que automatizam partes do processo, criando e mantendo vários artefactos, com ênfase nos modelos. O processo definido pode ser descrito graficamente segundo duas dimensões: uma horizontal, que define o tempo através de quatro fases de desenvolvimento distintas (preparação, elaboração, construção, transição), e uma vertical, que identifica as actividades que decorrem em simultâneo e os papéis (*roles*) que lhes estão adstritos.

O RUP está baseado num conjunto de nove actividades, que posicionam todos os trabalhadores e sub actividades em grupos lógicos. Para cada um desses processos é descrito quem deve fazer o quê, como e quando, ou seja, o que é que deve ser produzido, as habilidades necessárias para o fazer e uma explicação, passo a passo, de como os objectivos específicos do desenvolvimento devem ser atingidos.

A integração do processo do *ReModeler* deve acontecer dentro de um determinado período do ciclo de vida de software. Em analogia à figura do RUP, estabeleceu-se esse período identificando as várias fases que o constituem, representadas por áreas rosas na Figura 34. O intervalo começa um pouco depois do início da fase de construção, mais precisamente quando já foi implementado algum caso de utilização e este já se encontra numa fase estável. Isto acontece porque, para usar o *ReModeler*, é necessário ter disponível o código fonte e o respectivo executável, do sistema a documentar. O período de utilização das funcionalidades da ferramenta estende-se até ao fim da fase de transição, altura em que se realizam várias tarefas relacionadas com a gestão de alterações, manutenção e testes.

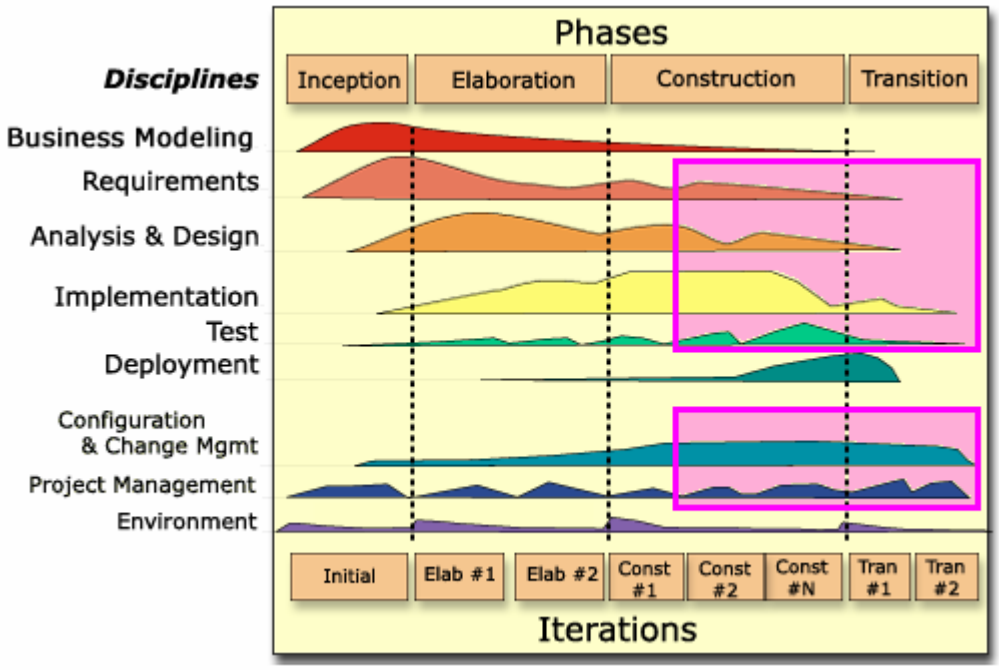


Figura 34. Identificação das fases do ciclo de vida onde se enquadram as funcionalidades do *ReModeler* (adaptado de [Rational, 2001]).

Capítulo 3

Arquitectura da solução

Conteúdo

Este capítulo descreve a arquitectura da solução proposta através dos seus componentes.

3 Arquitectura da solução

3.1 Apresentação da arquitectura do *ReModeler*

As técnicas de Modelação Aumentada e processo respeitante à sua aplicação, que são propostas no âmbito desta dissertação, são reificadas na ferramenta *ReModeler*, para aplicações escritas na linguagem Java. Esta ferramenta está a ser desenvolvida em colaboração com outro colega, que trata de conceitos visuais e de sincronização, face às capturas executadas [Gouveia, 2008]. Para melhor definir as fronteiras entre os trabalhos, desenvolvidos nas duas dissertações, foi criado um diagrama de componentes colorido. Os elementos com cor de fundo azul fazem parte do trabalho descrito em [Gouveia, 2008] e os de cor rosa fazem parte dos conceitos descritos nesta dissertação (ver Figura 36). É possível ainda ver elementos com o fundo de cor verde, que representam elementos exteriores ao sistema.

O diagrama está organizado segundo um componente geral, o *ReModeler*, que constitui o protótipo. Se abstrairmos a composição do *ReModeler*, podemos observar o sistema como uma caixa preta, apenas visualizando os componentes de entrada e necessários ao funcionamento do processo e os componentes que resultam das acções executadas pelo sistema (Figura 35). De entre os componentes de saída, podemos identificar todos os artefactos já descritos nesta dissertação, como os vários tipos de matrizes CRUD, os cartões CRC e os vários diagramas gerados. Para além destes, podemos ainda ver mais dois resultados, que são o filme do cenário e o manual do sistema [Gouveia, 2008].

A ferramenta *ReModeler* é constituída por um conjunto de componentes que tratam de cada funcionalidade separadamente: *Use Case Documenter*, *ReModeler Database*, *Test Navigator* (constituído por *Test Editor* e *TestCase Coverage*), *Scenario Capturer* (composto por *Internal Scenario Capturer* e *External Scenario Capturer*), *Interation Filter*, *Sequence Diagram Generator*, *Visual Scenario Player*, *Visual Scenario Generator*, *Sequence Diagram Animator*, *Requirement Implementation CRUD Matrices Generator*, *Colored Diagrams Generator*, *Extended CRC Cards Generator*, *Manual Player* e *Manual Generator*. De seguida passo a explicar os vários componentes (rosa) individualmente. Em [Gouveia, 2008] podem ser encontradas mais informações sobre os componentes de cor azul.

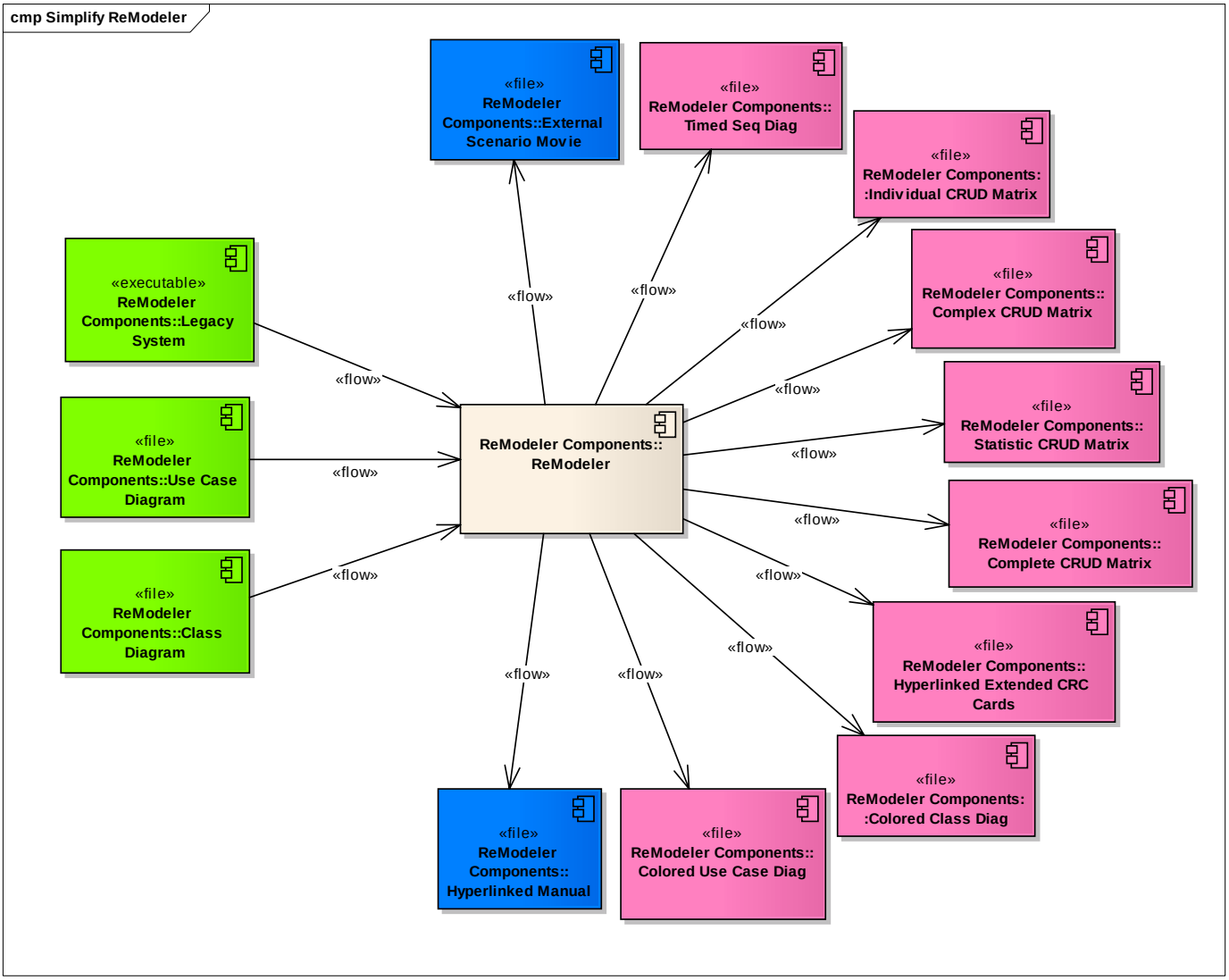


Figura 35. *ReModeler* como uma caixa preta

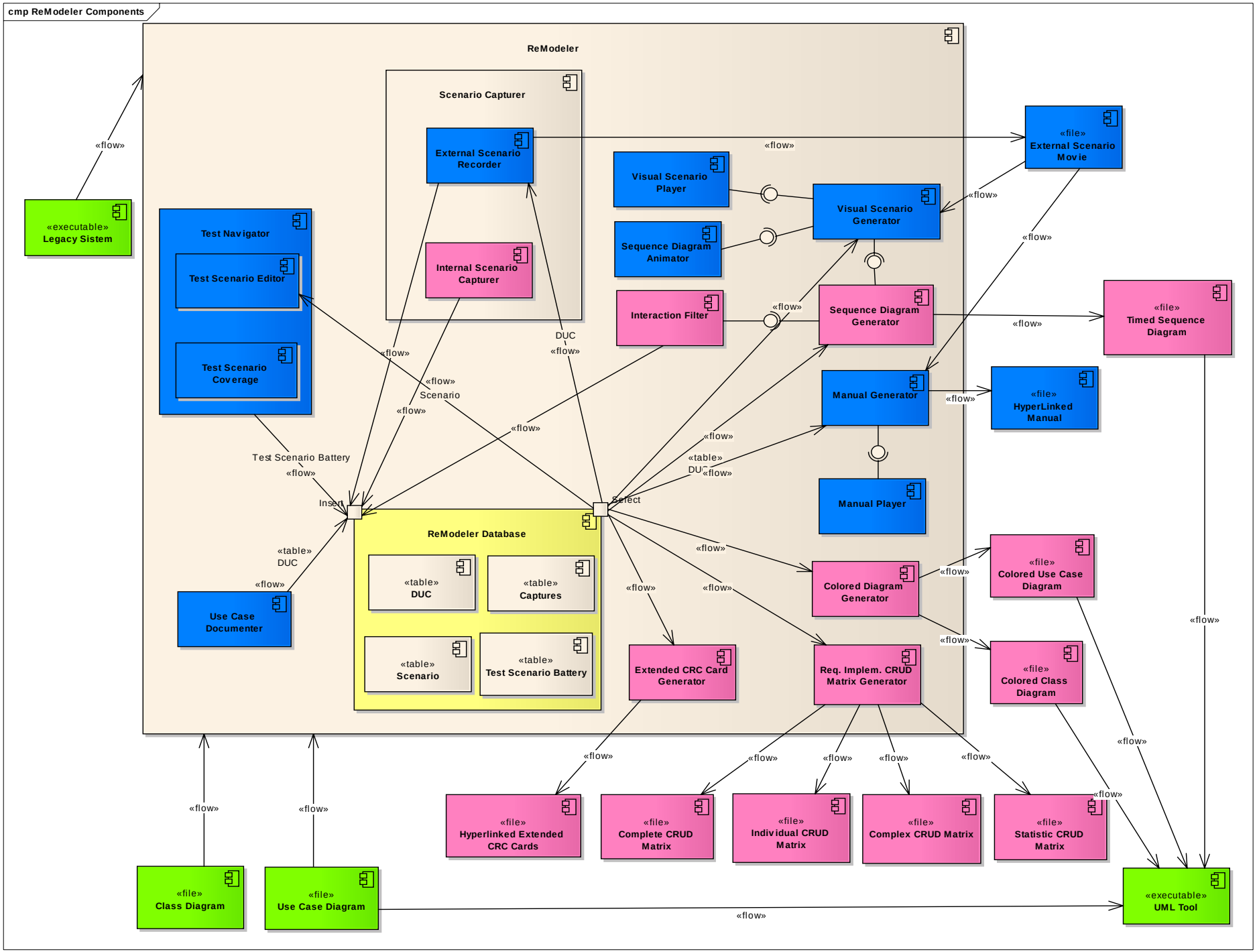

cmp ReModeler Components
ReModeler
Scenario Capturer
External Scenario Recorder
Internal Scenario Capturer
«executable» Legacy Sistem
Test Navigator
Test Scenario Editor
Test Scenario Coverage
Test Scenario Battery
Use Case Documenter
ReModeler Database
«table» DUC
«table» Captures
«table» Scenario
«table» Test Scenario Battery
Visual Scenario Player
Sequence Diagram Animator
Interaction Filter
Visual Scenario Generator
Sequence Diagram Generator
Manual Generator
Manual Player
Colored Diagram Generator
Extended CRC Card Generator
Req. Implem. CRUD Matrix Generator
«file» External Scenario Movie
«file» Timed Sequence Diagram
«file» HyperLinked Manual
«file» Colored Use Case Diagram
«file» Colored Class Diagram
«file» Hyperlinked Extended CRC Cards
«file» Complete CRUD Matrix
«file» Individual CRUD Matrix
«file» Complex CRUD Matrix
«file» Statistic CRUD Matrix
«file» Class Diagram
«file» Use Case Diagram
«executable» UML Tool
«flow»
Scenario
DUC
Insert
Select
«table» DUC

Figura 36. Componentes do *ReModeler.*

3.1.1 *ReModeler Database*

O componente *ReModeler Database* representa uma base de dados que estabelece uma ligação entre os restantes componentes no sistema. Esta é a razão de este ser o único componente com a cor amarela no diagrama. Qualquer informação que seja introduzida no sistema, directamente ou através de importação, é inserida nesta base de dados. Do mesmo modo, a captura dinâmica do sistema em análise gera um grande volume de informação, que é directamente armazenada na mesma forma persistente. Em qualquer das funcionalidades previstas na ferramenta, os dados da base de dados são lidos, são tratados e, se for o caso, são produzidos artefactos a partir dessa informação persistente. Isto permite não só uma gestão eficiente da memória, mas também o desacoplamento entre os vários componentes. Como a informação fica armazenada de forma persistente, torna-se possível a realização de várias sessões sem perdas de informação e a análise de vários sistemas em simultâneo. A estrutura da base de dados deste componente está explicada em detalhe no capítulo seguinte.

3.1.2 *Scenario Capturer*

Este componente é constituído por dois sub-componentes: o *Internal Scenario Capturer* e o *External Scenario Capturer*. Nesta dissertação apenas se fala do primeiro componente.

O *Internal Scenario Capturer* representa o componente onde é feita a captura dinâmica do sistema, mais precisamente do comportamento interno do sistema. Neste componente é feita a instrumentação do código e são capturadas as mensagens trocadas entre os vários objectos que constituem o sistema, aquando da execução de um cenário. Não existe nenhum entregável resultante da execução deste componente. As informações aqui recolhidas são directamente armazenadas na base de dados do *ReModeler* para posterior tratamento.

Este componente é implementado à custa de um conjunto de pacotes. O diagrama da Figura 37 mostra os pacotes que participam nessa implementação. O

principal pacote deste componente é o *Capture*. É nele que se executam todas as acções referidas anteriormente. Tanto o pacote de *UML*, como o pacote de *Control*, são pacotes de apoio. O pacote de *Control* está presente em todos os componentes e ajuda a estabelecer bases de apoio, como informações sobre o sistema em análise. A apresentação de cada um dos pacotes e respectivas classes constituinte é feita no capítulo seguinte (“Desenho da Solução”).

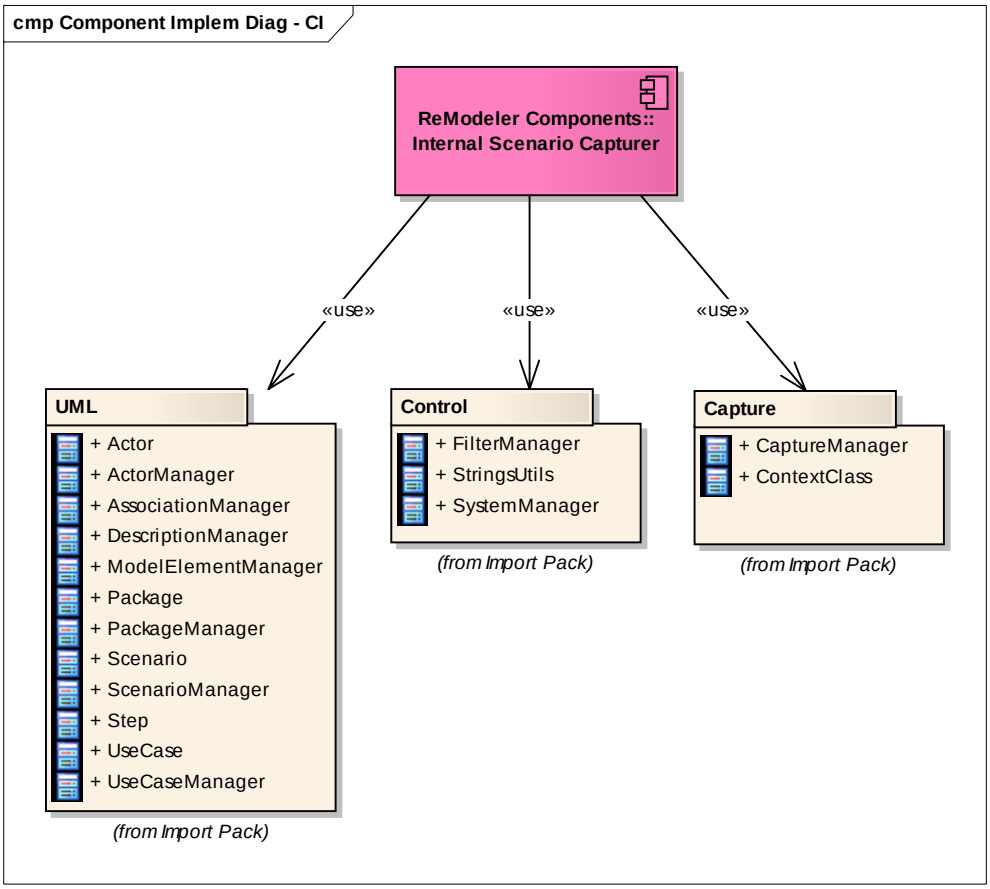


Figura 37. Implementação de Componentes – *Internal Scenario Capturer*.

3.1.3 *Interaction Filter*

O componente *Interaction Filter* trata da filtragem das mensagens. Quando é feita uma captura de um cenário em execução, obtém-se um grande volume de informação. A análise dessa informação pode tornar-se desnecessária e às vezes impraticável. A possibilidade de filtrar parte da informação recolhida é um mecanismo de vital importância para os analistas.

Neste componente é dada a possibilidade de escolher dois tipos de filtragem: (i) a filtragem de classes que participam numa execução ou (ii) a filtragem de classes

do sistema (sem necessidade de prévia execução). O analista escolhe apenas a informação que é relevante para ele, podendo realizar várias filtragens relativas a uma captura, conseguindo várias visões parciais da mesma. Este componente está implementado por um pacote principal que é o pacote de *Control* e por um de apoio, *CRUD*, como mostrado na Figura 38.

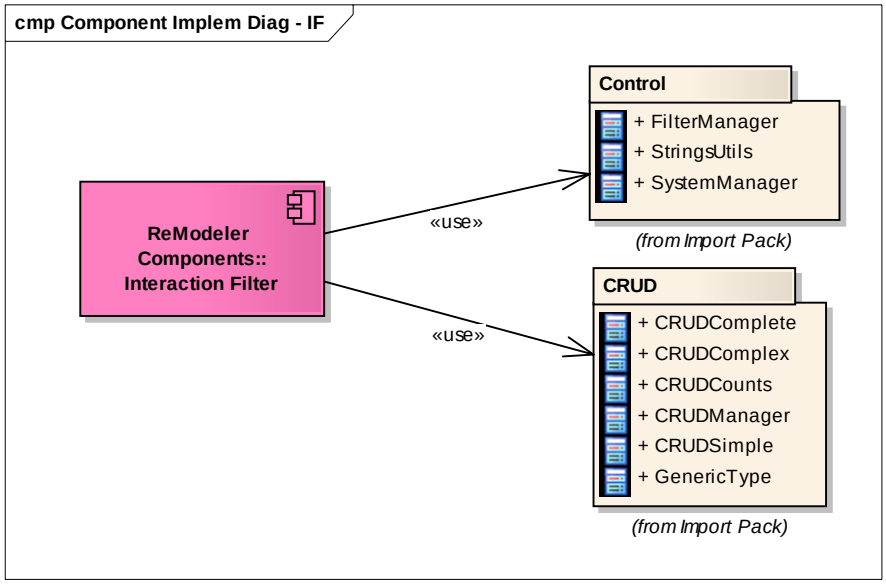


Figura 38. Implementação de Componentes – Interaction Filter

3.1.4 *Sequence Diagram Generator*

Este componente tem como funcionalidade principal gerar diagramas de sequência da UML.

Existem dois tipos de resultados (diagramas) produzidos por este componente, um é interno e o outro é exportado como artefacto do *ReModeler*. Os diagramas internos ao *ReModeler* têm um formato próprio e são passados ao componente *Visual Scenario Player* para serem animados e posteriormente visualizados [Gouveia, 2008]. Os diagramas de sequência externos são os que são exportados em formato XMI, para posterior importação numa ferramenta UML, e que estão representados na Figura 36 como *Timed Sequence Diagram*.

Para qualquer das duas gerações possíveis, o componente tem de ler as informações que foram armazenadas na base de dados aquando da captura do cenário, e os indicadores de filtragem, porque são eles que vão definir o que vai ser representado em cada diagrama.

Na implementação deste componente estão envolvidos três pacotes, como mostra a Figura 39. O pacote *XMI* é onde ocorrem as principais acções de leitura e tratamento dos dados e onde são gerados os diagramas. Os pacotes *Control* e *Capture* apenas servem para apoiar algumas destas actividades.

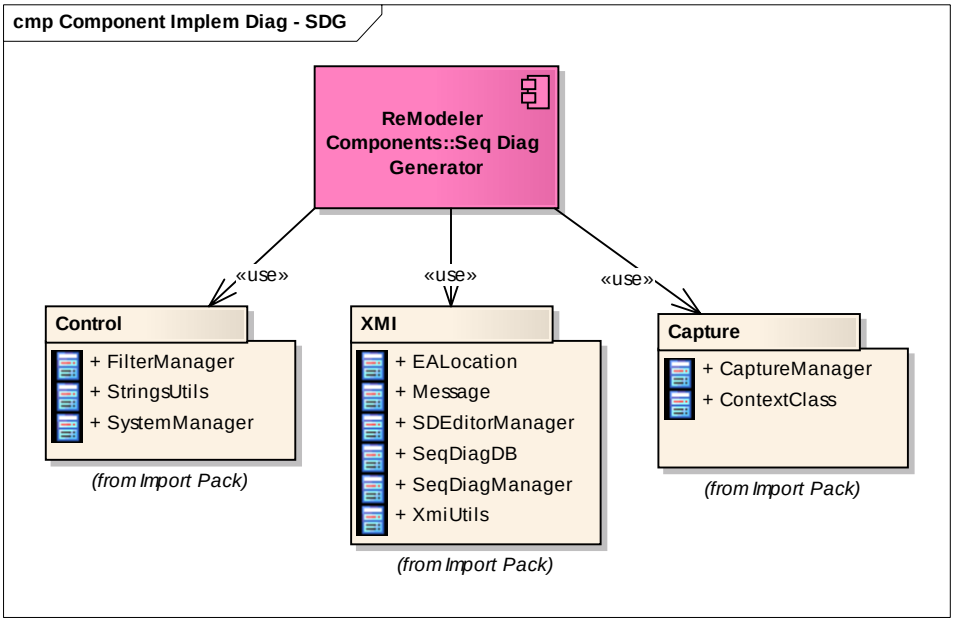


Figura 39. Implementação de Componentes – *Sequence Diagram Generator.*

3.1.5 *Colored Diagram Generator*

É no componente *Colored Diagram Generator* que são tratados e gerados os diagramas UML coloridos. Este componente produz dois resultados exteriores, representados na Figura 36, que são o diagrama de casos de utilização colorido (*Colored Use Case Diagram*) e o diagrama de classes colorido (*Colored Class Diagram*). Estes diagramas coloridos reflectem a cobertura de capturas do sistema, no primeiro caso, e a cobertura ou intensidade de execução da estrutura do sistema, no outro caso. Ambos os diagramas são exportados em formato XMI, para permitir a sua visualização numa ferramenta de UML independente. A análise da cobertura é feita a partir da leitura das informações estáticas, importadas para o sistema, e sua comparação com as informações dinâmicas recolhidas em cada captura, armazenadas na base de dados. Para além disso, no caso da geração dos diagramas de classes é dada a possibilidade ao utilizador de definir a palete de cores e o intervalo de percentagem que corresponde a cada cor.

A implementação deste componente é feita através de quatro pacotes, como mostra a Figura 40. O pacote primordial às funcionalidades deste componente é o

pacote *Color*. O pacote *ReModelerInterface* permite a interacção com o utilizador, nomeadamente na definição dos intervalos de cores. Os restantes dois pacotes, *UML* e *Control* apoiam a execução das funcionalidades.

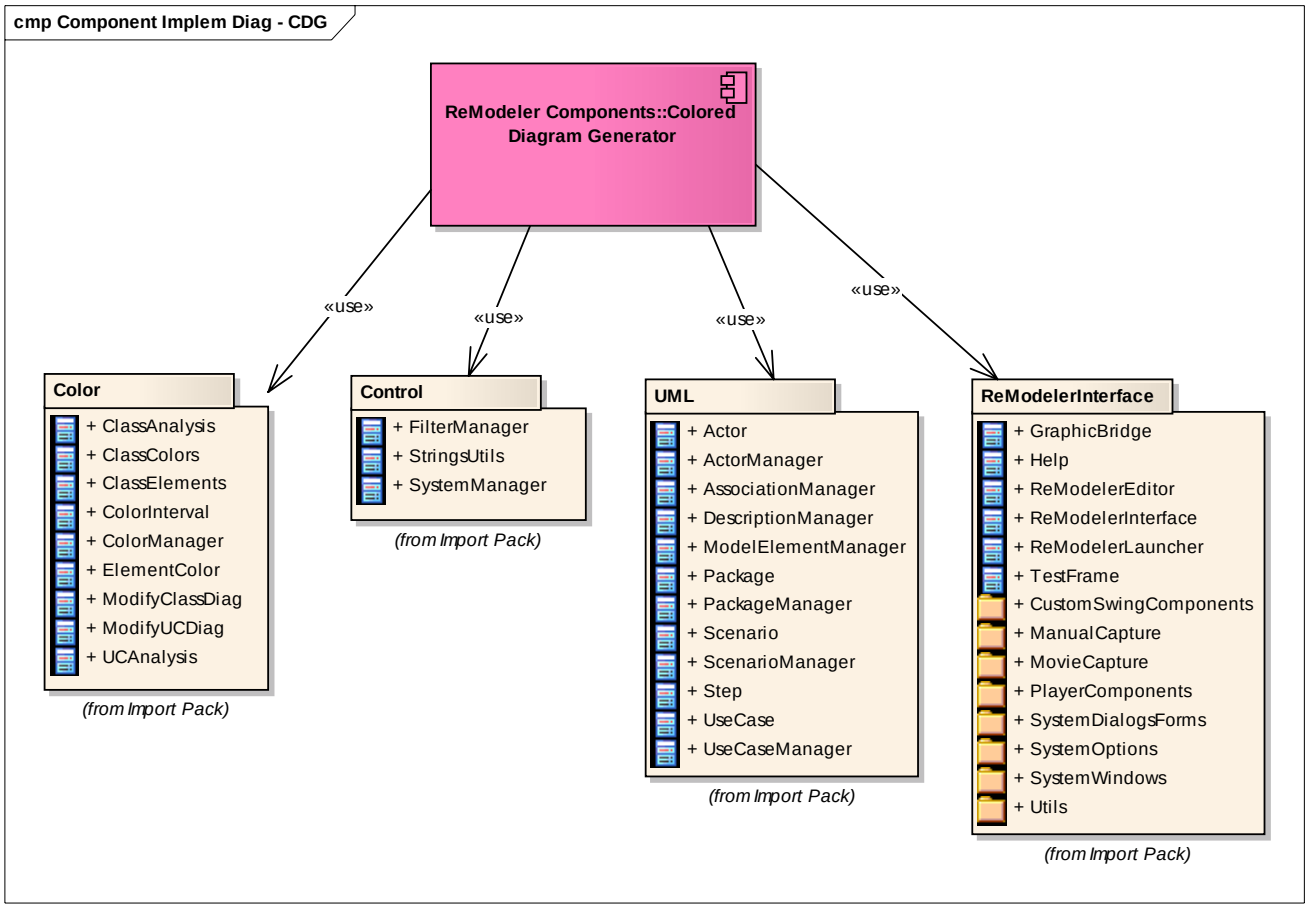


Figura 40. Implementação de Componentes – *Colored Diagram Generator.*

3.1.6 *Requirement Implementation CRUD Matrices Generator*

Requirement Implementation CRUD Matrices Generator é o componente que permite gerar os vários tipos de matrizes CRUD para um sistema. Este é o componente que produz mais resultados exteriores, como mostra a Figura 36. São possíveis de ser gerados quatro tipos de matrizes: (i) a matriz simples, individual para uma execução de cenário, (ii) a matriz completa, que mostra a informação para todas as capturas feitas no sistema, (iii) a matriz complexa, que permite ao utilizador escolher a granularidade a que quer ver as informações, e (iv) a matriz estatística, que adiciona informação à matriz original contendo o número de operações que ocorreram.

As várias matrizes são produzidas através da leitura e análise das informações guardadas na base de dados no momento das capturas. Em cada uma, embora de uma forma distinta, as informações são tratadas e é produzido o ficheiro em *HTML* respectivo. No caso da matriz complexa torna-se ainda necessário permitir ao utilizador a escolha da granularidade pretendida, através da mostragem da estrutura interna do sistema.

Da Figura 41 podemos ver que este componente foi implementado por quatro pacotes. O pacote principal é o pacote com o nome *CRUD*. Aqui são criadas todas as matrizes consoante o tipo pretendido. Mais uma vez, os restantes pacotes, *CRC*, *UML* e *Control*, são usados na implementação do pacote principal.

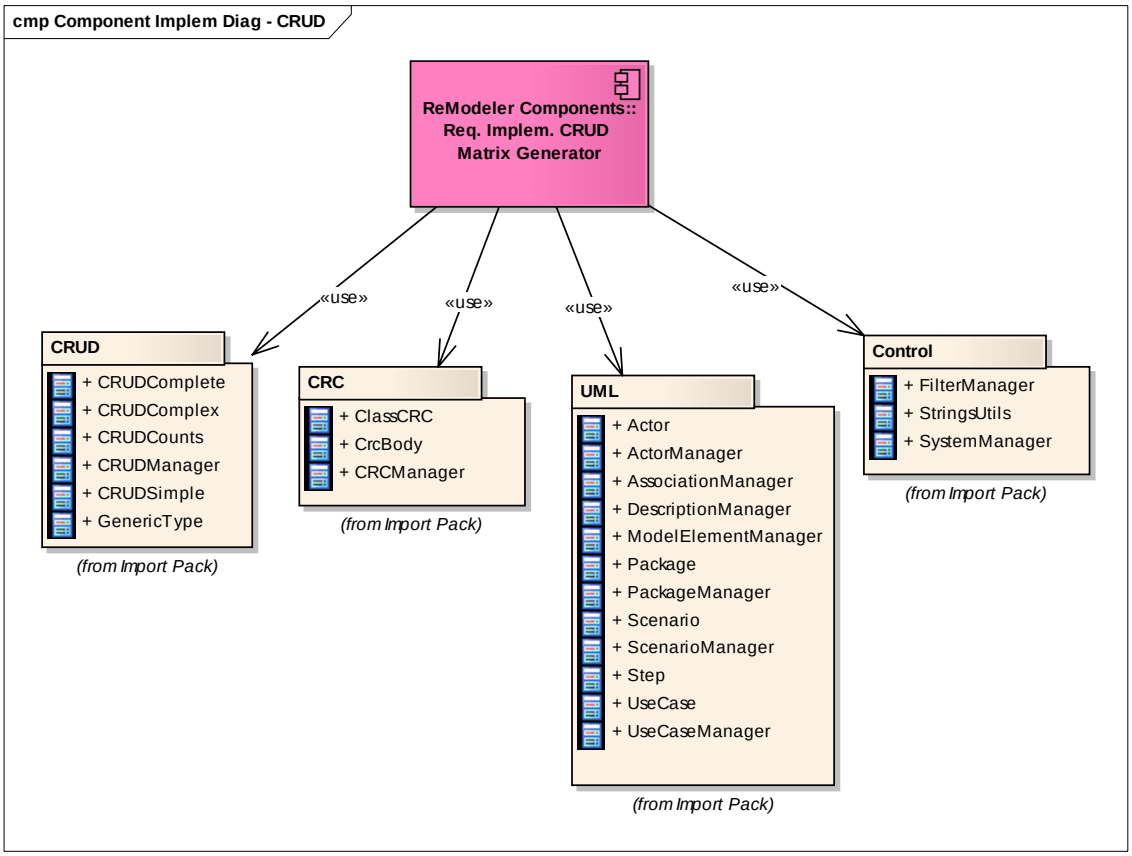


Figura 41. Implementação de Componentes – *Req. Implem. CRUD Matrix Generator.*

3.1.7 *Extended CRC Card Generator*

O componente *Extended CRC Card Generator* gera os cartões CRC para um sistema. Cada cartão CRC é produzido através da análise e tratamento dos dados lidos da base de dados, respeitantes às capturas dos cenários do sistema. O resultado

deste componente é conjunto dos cartões em formato *HTML*, como mostra a Figura 36.

Este componente está implementado por três pacotes, o *CRC*, o *Control* e o *UML*, como mostrado na Figura 42. Como o nome indica, o pacote que tem a função de gestão e geração dos cartões é o pacote *CRC*. Tanto o pacote *Control*, como o pacote *UML*, disponibilizam funcionalidades de apoio à execução do pacote principal.

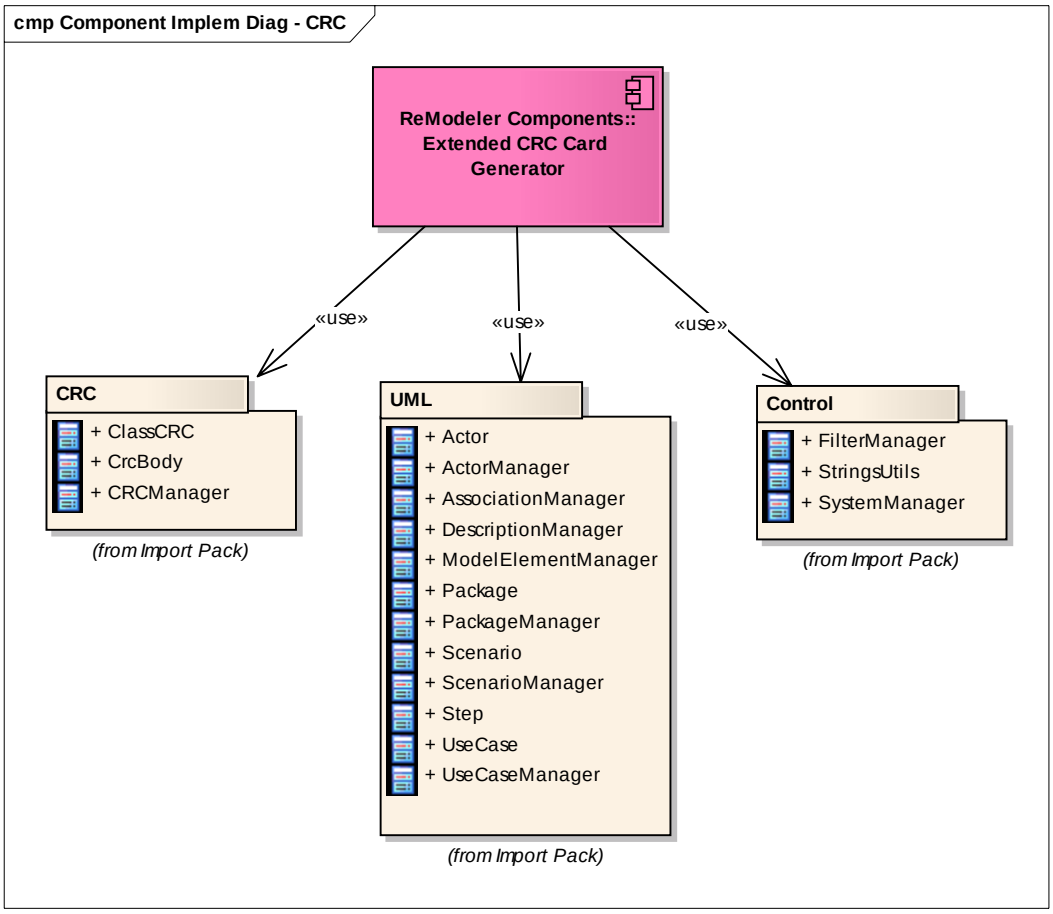


Figura 42. Implementação de Componentes – *Extended CRC Card Generator*.

Capítulo 4

Desenho Detalhado

Conteúdo

Neste capítulo é detalhada a solução proposta, descrevendo os aspectos relacionados com a implementação do ReModeler.

4 Desenho Detalhado

4.1 Introdução Geral

No capítulo anterior foi possível analisar os vários componentes que constituem o sistema *ReModeler* e os pacotes que implementam cada componente. Existe um componente central que funciona como meio de ligação entre todos os outros, que é a base de dados do *ReModeler*. A base de dados do *ReModeler* guarda a informação recolhida em cada captura, para posteriormente ser lida e permitir gerar os vários artefactos propostos. Do seu bom desenho e desempenho depende o funcionamento dos restantes componentes.

Neste capítulo é descrito o desenho e implementação da base de dados, assim como o dos restantes componentes detalhados por pacotes.

4.2 Base de Dados do *ReModeler*

Quando é feita uma captura de um cenário de um sistema, o volume de dados recolhido pode ser muito grande, mesmo para sistemas pequenos (ver capítulo 5). Esta grande quantidade de dados, se ficasse guardada em memória transiente, iria comprometer seriamente a performance, quer do *ReModeler*, quer do sistema em análise. Esta preocupação veio originar a necessidade de criar um mecanismo de armazenar de forma persistente esses mesmos dados, ou seja, a criação da base de dados do *ReModeler*.

A base de dados do *ReModeler* foi inicialmente pensada apenas para armazenar os dados provenientes de cada captura do sistema. No entanto, com a análise do resto do sistema tornou-se claro que a sua utilização poderia ser mais abrangente. Toda a informação com a qual o sistema tem de lidar passou a ser candidata a estar guardada na base de dados. O grande volume de dados e os vários tipos diferentes de informação a armazenar, fez com que se criasse um esquema da base de dados, com preocupações de eficiência e organização da informação, através de um modelo sob a forma de um diagrama de classes da UML, em que cada classe ficou com o estereótipo de tabela. O modelo está dividido em dois sub pacotes: o *Element Model* e o *Scenario Capturer*.

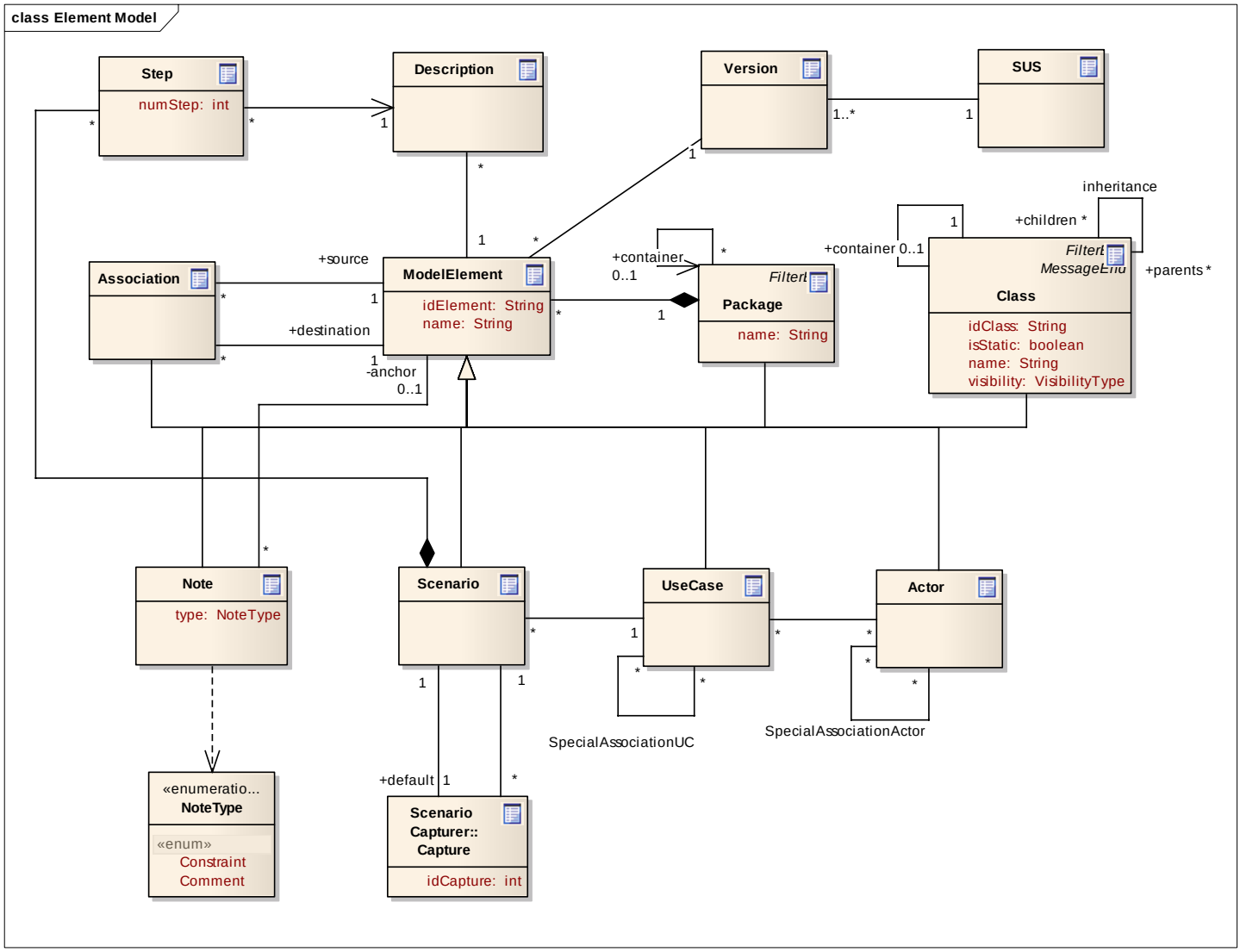


Figura 43. Modelo da Base de dados – sub pacote *Element Model*.

No primeiro sub pacote estão representados os elementos que constituem a base das funcionalidades do *ReModeler*, e que podem ser encontrados em vários diagramas da UML, como está mostrado na Figura 43. De entre os seus constituintes podem ser vistas classes como a que representa o sistema em análise (SUS – *System Under Study*), a que armazena as suas versões e ainda uma classe que funciona como elemento base dos restantes, o *ModelElement*. Todos os outros elementos deste sub pacote, como os casos de utilização, os cenários, as classes, os pacotes, os actores, as associações, etc, vão apresentar-se como generalizações de *ModelElement*. Basicamente, um sistema, que pode ter várias versões, vai ter associado um conjunto de *ModelElements* que o vão caracterizar, frutos da aplicação do processo proposto. Os actores do sistema, os casos de utilização criados, assim como os respectivos

cenários, vão ser armazenados, nas classes do mesmo nome. As descrições detalhadas dos cenários do sistema podem ser introduzidas através de instâncias da classe *Step*, que se unem no conceito de descrição. Qualquer elemento da modelação deverá estar definido do âmbito de um pacote. Na verdade, o mesmo já acontece em muitas ferramentas de modelação UML que criam à partida um pacote, onde todos os elementos e diagramas vão sendo adicionados. Ao nível da implementação, este fenómeno é normalmente encontrado no conceito de pacotes que aglomeram um conjunto de classes, como os pacotes de Java.

No segundo sub pacote estão representados os elementos que estão de algum modo envolvidos com a captura de cenários, como mostra a Figura 44. Na captura de execução de um sistema, relativamente a um cenário, são recolhidas as mensagens, com todas as suas características como o seu nome, argumentos e tipo de retorno, que são trocadas entre os vários objectos do sistema. Esta informação é necessária para determinar qual é a operação correspondente, isto é, aquela da qual a mensagem é uma invocação, desambiguando assim possíveis situações de sobrecarga de operações (operações com o mesmo nome, mas parâmetros distintos, seja em tipo ou em número). Neste sub pacote podemos encontrar elementos relativos à organização das capturas no sistema, ao armazenamento dos dados provenientes dessas capturas e da importação de diagramas de classes, e ainda ao conceito de filtragem de informação para visualização num diagrama. Para o primeiro conjunto de elementos, destacam-se classes como a *Capture*, que representa as capturas de são executadas para cada cenário do sistema, a classe *Message*, onde estão armazenadas as mensagens que foram trocadas dentro de cada captura, assim como a identificação de qual a fonte e o destino da mesma, através da classe *MessageEnd*. Esta última pode representar uma classe (no caso de invocação estática) ou uma instância, objecto, de uma classe. No segundo conjunto de elementos é possível referir a classe *Atribute*, que representa os atributos que pertencem a uma classe, apenas conseguidos pela importação de diagramas, a classe *Operation*, que descreve a operação que foi invocada ou importada e ainda as classes *Parameter* e *Argument*, que diferem entre si pela forma como são instanciadas: a primeira apenas o é através da importação, enquanto que a segunda é instanciada aquando das capturas, com os valores reais. O último dos conjuntos de elementos trata do armazenamento das informações relativas aos filtros (vide capítulo 2), como se pode ver pelas classes *Filter* e *FilterEnd*. Um *FilterEnd* representa qualquer elemento que pode ser filtrado, seja ele um pacote,

classe ou método, enquanto que um *Filter* é o agrupamento de todos os *FilterEnds* escolhidos num dado momento. Os filtros ficam armazenados e podem ser reutilizados sempre que forem necessários.

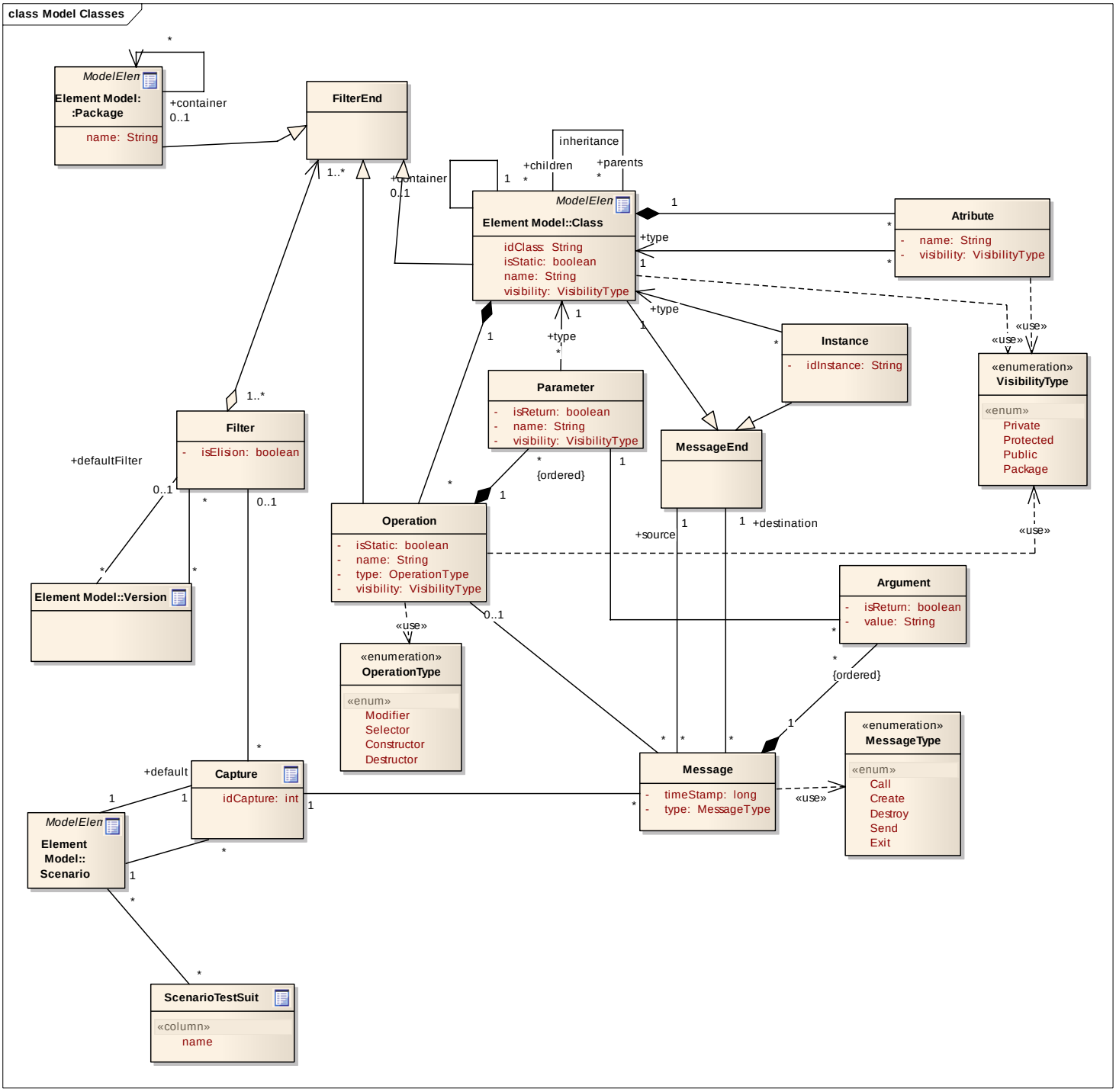


Figura 44. Modelo da Base de dados – sub pacote *Scenario Capturer*.

Finalizado o modelo da base de dados, foi usada uma facilidade de *forward engineering* da ferramenta UML usada (*Enterprise Architect*), que permitiu gerar automaticamente os ficheiros de criação dos elementos da base de dados (*Data Definition Language - DDLs*), a partir dos diagramas de classes atrás discutidos.

A base de dados do *ReModeler* foi implementada em *PostgreSQL* [PostgreSQL]. O *PostgreSQL* é um sistema de gestão de bases de dados relacional (SGBD) que vem a ser desenvolvido desde há 15 anos, livre e compatível com a maior parte dos sistemas operativos. A versão actual do *PostgreSQL* tem interfaces de utilização para bastantes linguagens nativas, como o Java, C/C++, etc. Ele inclui a maioria dos tipos de dados presentes no SQL92 e no SQL99, e implementa outras facilidades como o reconhecimento do conceito de herança (já existente no SQL99).

```
CREATE TABLE Class (
        idClass varchar(256) NULL,
        isStatic boolean NULL,
        name varchar(128) NULL,
        visibility VisibilityType NULL,
        classParent integer NULL,
        classID serial NOT NULL
) INHERITS (ModelElement,MessageEnd,FilterEnd);

ALTER TABLE Class ADD CONSTRAINT PK_Class
        PRIMARY KEY (classID);

ALTER TABLE Class ADD CONSTRAINT FK_Class_Class
        FOREIGN KEY (classID) REFERENCES Class (classID);

ALTER TABLE Class ADD CONSTRAINT FK_Class_ClassParent
        FOREIGN KEY (classParent) REFERENCES Class (classID);
```

Figura 45. Código SQL para a criação de uma tabela com implementação do conceito de herança.

A geração automática das DDLs apresentou no entanto algumas limitações. O reconhecimento do conceito de herança, representado nos diagramas, não foi totalmente satisfeito, o que exigiu a sua completude manual. Por outro lado, embora as classes tivessem preparadas para terem chaves únicas (atributo *idClass* da Figura 45), o gerador da ferramenta não conseguiu identificar potenciais atributos candidatos a satisfazerem esse requisito e originou a geração de chaves únicas automaticamente, como é possível ver na Figura 45, relativamente ao atributo *classID*. Posteriormente, o atributo *idClass* foi reaproveitado internamente, para atribuição de um identificador não numérico para efeitos de exportação.

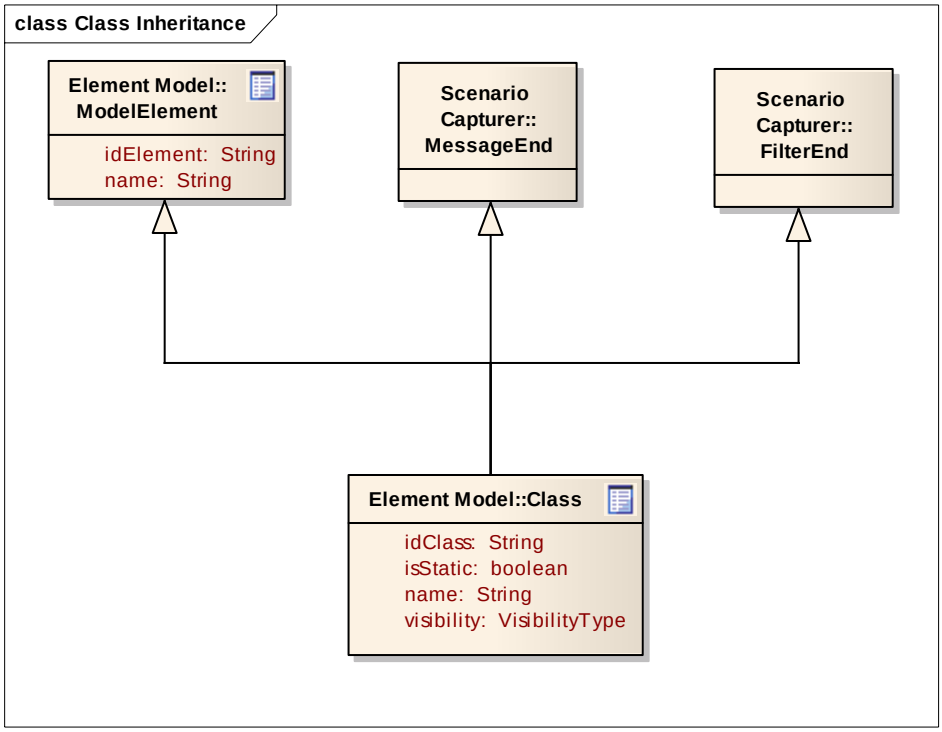


Figura 46. Excerto do diagrama, com herança múltipla.

Quando os ficheiros gerados são carregados na base de dados, as tabelas com herança comportam-se do mesmo modo que duas classes herdadas, ou seja, os atributos da tabela pai passam a estar também presentes em cada tabela filha. Mais precisamente, para o exemplo da tabela *Class*, que implementa múltipla herança, representado pelo excerto do diagrama da Figura 46, foi criado o excerto do ficheiro DDL, mostrado na Figura 45, que quando carregado na base de dados, adiciona comentários à tabela criada referentes aos atributos herdados, ver Figura 47.

A múltipla herança traz um problema adicional que se prende com a identificação dos atributos herdados, quando existe convergência dos mesmos. Se existir um atributo com o mesmo nome e tipo, coincidente em alguma das tabelas *ModelElement*, *MessageEnd* ou *FilterEnd*, o *PostgreSQL* não consegue fazer uma renomeação eficaz, na tabela *Class*, limitando-se a ignorar uma das ocorrências desse atributo. Isto originou a que houvesse uma preocupação adicional em criar atributos com nomes únicos, para tabelas que implementassem herança, para evitar perda de informação.

```
-- Table: "class"
-- DROP TABLE "class";

CREATE TABLE "class"
(
-- Herdada:   idelement character varying (128),
-- Herdada:   "name" character varying (128),
-- Herdada:   modelelementid integer NOT NULL DEFAULT
nextval('modelelement_modelelementid_seq'::regclass),
-- Herdada:   packageid integer,
-- Herdada:   versionid integer,
-- Herdada:   messageendid integer NOT NULL DEFAULT
nextval('messageend_messageendid_seq'::regclass),
-- Herdada:   filterendid integer NOT NULL DEFAULT nextval('filterend_filterendid_seq'::regclass),
  idclass character varying(256),
  isstatic boolean,
  visibility visibilitytype,
  classparent integer,
  classid serial NOT NULL,
  (…)
) INHERITS (modelelement, messageend, filterend);
```

Figura 47. Implementação de uma tabela com atributos herdados.

Todas as funções que inserem, modificam, apagam ou lêem elementos da base de dados foram criadas e mantidas dentro da própria base de dados, em *stored procedures*. Um *stored procedure* é um conjunto de instruções SQL que pode ser mantido na base de dados, para que os clientes não tenham que reescrever as instruções repetidamente sempre que pretendem obter o mesmo resultado. Para além disso, os *stored procedures* podem melhorar grandemente a performance da aplicação, não só por limitarem o tráfego de informação entre a aplicação cliente e o SGBD, mas também por pré compilarem as funções inicialmente, o que facilita bastante as invocações repetitivas. Algumas dessas funções, nomeadamente as que apagam elementos, requereram *triggers* para garantir a fiabilidade dos dados. Na Figura 48 é possível ver dois exemplos de funções criadas. Estas funções foram todas reproduzidas para um ficheiro de *script*, para permitir o carregamento da base de dados sempre que necessário.

```
-- Function: "getCountScenarioModifier"(integer)
-- DROP FUNCTION "getCountScenarioModifier"(integer);

CREATE OR REPLACE FUNCTION "getCountScenarioModifier"(scenarioid integer)
  RETURNS bigint AS
$BODY$
select count(operation.operationid) from message, operation, capture, scenario
where
        message.operationid = operation.operationid and
        message.captureid = capture.captureid and
        scenario.defaultsc = capture.captureid and
        operation.type = 'Modifier'and
        scenario.scenarioid = $1;
$BODY$
  LANGUAGE 'sql';
ALTER FUNCTION "getCountScenarioModifier"(integer) OWNER TO postgres;
```

```
-- Function: deleteMessage()
-- DROP FUNCTION deleteMessage();

CREATE OR REPLACE FUNCTION deleteMessage()
  RETURNS trigger AS
$BODY$
BEGIN
  delete from argument where argument.messageid = OLD.messageid;
  RETURN old;
END
$BODY$
  LANGUAGE 'plpgsql' ;
ALTER FUNCTION deleteMessage() OWNER TO postgres;

CREATE TRIGGER deleteMessArg BEFORE DELETE ON message
   FOR EACH ROW EXECUTE PROCEDURE deleteMessage();
```

Figura 48. Implementação de duas funções em SQL.

4.3 Implementação do *ReModeler*

A implementação das funcionalidades presentes no *ReModeler* foi realizada em Java e *AspectJ* [Eclipse, 2008] (ver Anexo A). A comunicação com a base de dados foi realizada através da interface de programação *Java Database Connectivity Application Program Interface* (JDBC API) [Sun Microsystems, 2008] (ver Anexo A).

Cada funcionalidade do *ReModeler* foi implementada por um pacote separado, existindo, como seria de esperar, algum acoplamento entre eles. A Figura 49 mostra como os pacotes estão organizados e como se relacionam, através de um diagrama de pacotes UML. Em seguida vão ser explicados e detalhados os pacotes que constituem esta arquitectura.

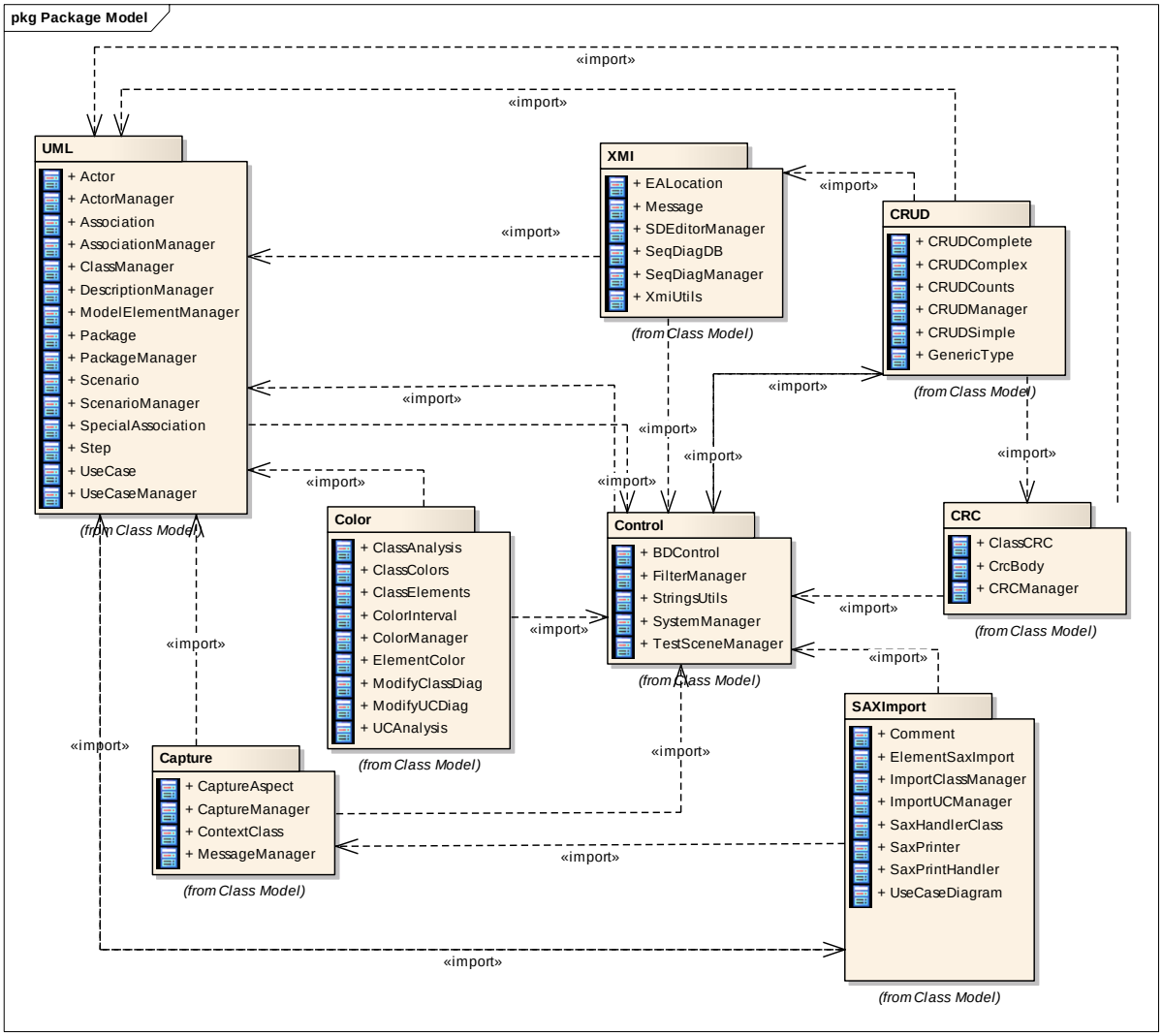


Figura 49. Diagrama de Pacotes para o sistema *ReModeler*

4.3.1 Pacote *Control*

Quando se pretende analisar um sistema, este tem de ser entrelaçado com o *ReModeler*. Para além disso, é necessárias levar a cabo um conjunto de acções que inicializam as estruturas internas, quer na base de dados, quer nos vários pacotes constituintes do *ReModeler*. O pacote *Control* é o pacote que trata de inicializar esses dados e variáveis que vão permitir o funcionamento do sistema. Este pacote é constituído por quatro classes que estão representadas no diagrama de classes da Figura 50.

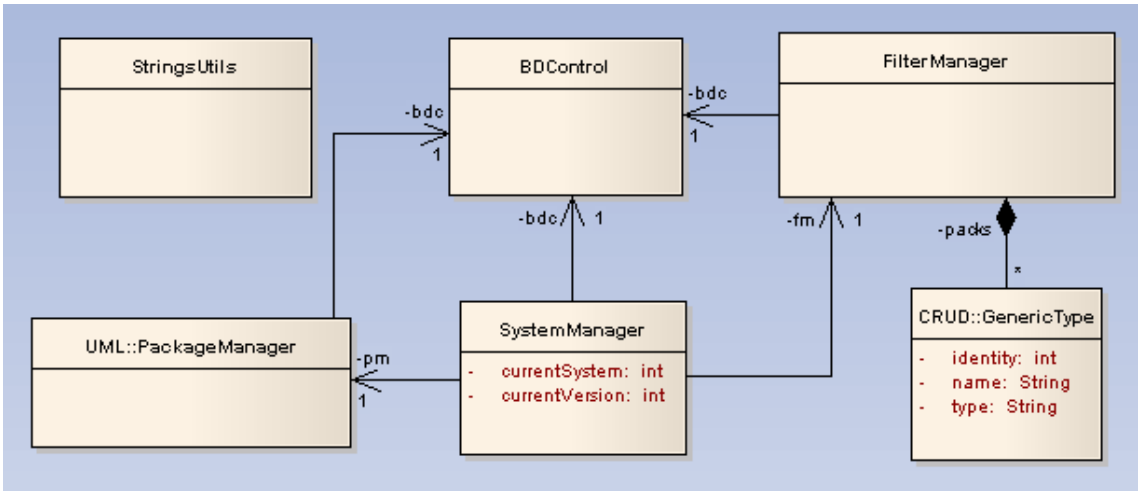


Figura 50. Diagrama de Classes para o pacote *Control*.

4.3.1.1 *StringsUtils*

Esta classe serve de apoio a várias funcionalidades dentro do *ReModeler*, disponibilizando um conjunto de métodos que permite tratar das *Strings* que vão aparecer em praticamente todos os artefactos gerados, como a preparação dos parâmetros para serem escritos em diagramas, a criação de identificadores únicos para identificar instâncias, etc.

4.3.1.2 *SystemManager*

Quando se começa a análise de um novo sistema, ou mesmo de um sistema já existente, é necessário proceder à introdução na base de dados de informações como o seu nome, a sua versão, um filtro e um pacote por defeito. É nesta classe *SystemManager* que essas acções tomam lugar.

Esta classe disponibiliza métodos que informam sobre os dados do sistema e da versão que está a ser correntemente utilizada, e por este motivo é uma das classes mais usadas. A identificação do sistema e respectiva versão em análise é uma forma de conseguir a unicidade dos dados da base de dados e garantir que as pesquisas e actualizações são feitas no contexto correcto.

4.3.1.3 *FilterManager*

Quando é realizada uma captura, pode ser necessário filtrar as mensagens que são trocadas entre os objectos do sistema. Cabe ao utilizador escolher as informações que pretende visualizar e analisar, filtrando-as por pacotes, classes ou métodos. Estas filtragens são inseridas no *ReModeler* e ficam armazenadas de modo persistente para posterior reutilização.

A classe que está incumbida de tratar das filtragens é a classe *FilterManager*. Esta classe disponibiliza um conjunto de métodos que permite inserir, ler e apagar filtros da base de dados. Para além disso, ela vai também disponibilizar métodos para ler e preparar os elementos para serem filtrados: pacotes, classes ou métodos.

4.3.1.4 *BDControl*

Quando se pretende comunicar com a base de dados, seja para pesquisar, actualizar, inserir ou apagar elementos, no contexto de uma sessão é necessário estabelecer uma ligação JDBC com a mesma. Esta classe disponibiliza dois métodos que servem para criar e fechar essas ligações, como mostra a Figura 51.

```
public Statement initDriverConnection() {
  try{
       Class.forName("org.postgresql.Driver");//LOAD THE DRIVER
       String url = "jdbc:postgresql://localhost:5432/ReModeler";//ESTAB.URL
       Connection db = DriverManager.getConnection(url,
             "postgres","quasarReModeler");//ESTAB. DB CONNECTION

       Statement stmt = db.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_READ_ONLY);
       stmt = db.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                        ResultSet.CONCUR_READ_ONLY);
       return stmt;
  } catch (Exception e) {e.printStackTrace();}      return null;
}

public void closeDriverConnection(Statement stmt) {
  try{
       stmt.getConnection().close();
       stmt.close();
  } catch (Exception e) {e.printStackTrace();}
}
```

Figura 51. Métodos de abertura e fecho de conexão JDBC.

Em qualquer classe que se pretenda invocar uma função da base de dados, o processo deve ser iniciado com a abertura da conexão, realizada no início de cada sessão, depois deve ser preparado o pedido e seguidamente executado. Quando a

invocação tem valores a retornar, deve ser criado um *ResultSet* que os guarda para posterior leitura e tratamento. Um *ResultSet* é um conjunto de linhas da base de dados, resultantes da execução de um *select*, e que podem ser processados através de cursores. No fim da sessão, deve ser usado o método que fecha a conexão inicialmente criada. Esta sequência de operações está esquematizada no diagrama de sequência da Figura 52.

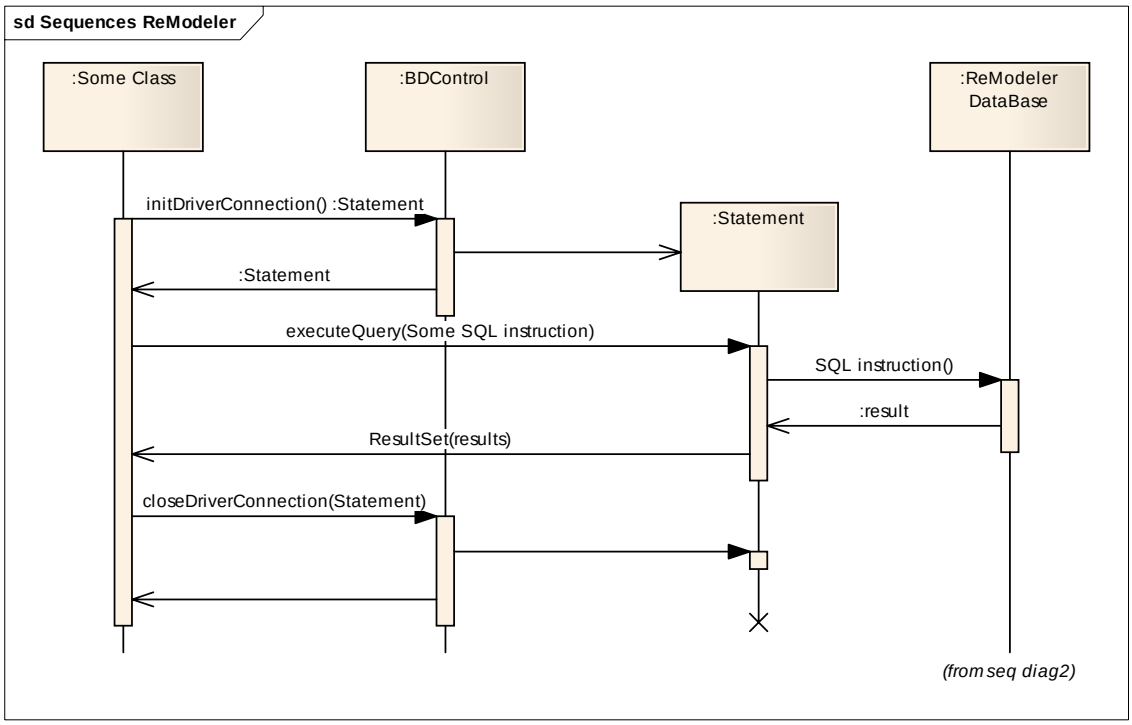


Figura 52. Diagrama de Sequência para a ligação com a base de dados.

4.3.1.5 *TestSceneManager*

O pacote de *Control* tem ainda uma última classe que é o *TestSceneManager*. Esta classe trata da gestão de agrupamentos das baterias de cenários de testes e a sua descrição, que é realizada em [Gouveia, 2008], cai fora do âmbito desta dissertação.

4.3.2 Pacote CRC

O pacote CRC é onde estão implementadas as funcionalidades relativas à geração do artefacto *Extended CRC Cards*. A criação e gestão dos cartões CRC estão implementadas por três classes: *CRCManager, Class CRC e CRCBody*, como mostra o diagrama da Figura 53. As restantes classes do diagrama são classes que auxiliam a

execução das funcionalidades, como as do pacote *Control* anteriormente faladas e as do pacote *UML*.

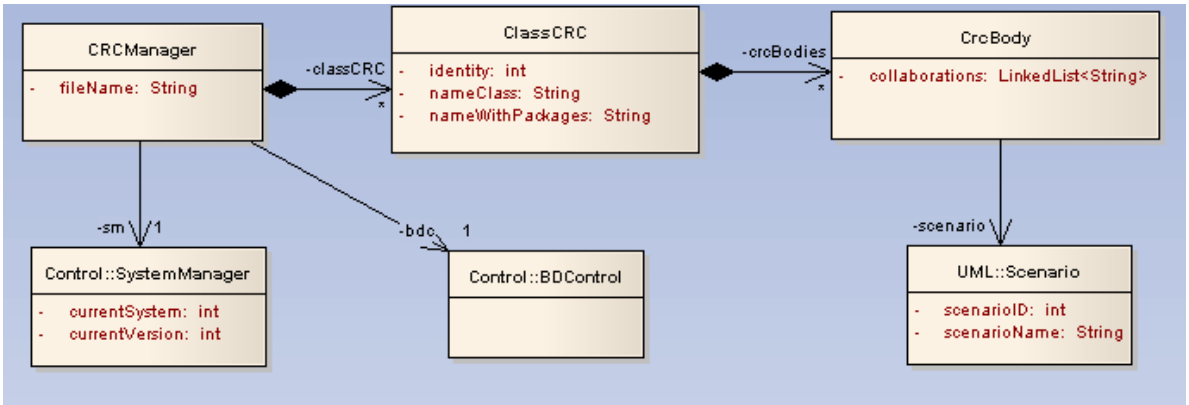


Figura 53. Diagrama de classes para o pacote CRC.

4.3.2.1 *Class CRC*

A classe *Class CRC* representa um cartão para uma determinada classe do sistema. Ela inclui todos os elementos que constituem um cartão CRC, como o nome da classe e o conjunto de responsabilidades e colaborações. Cada cartão CRC refere um conjunto de cenários (as responsabilidades), em que está envolvida a classe respectiva. Para cada um desses cenários, sabem-se quais as outras classes envolvidas. Cada responsabilidade é representada pela classe *CRC Body*.

4.3.2.2 *CRC Body*

Aqui está representado o que normalmente é encontrado no corpo de um cartão CRC de uma classe, ou seja, as suas responsabilidades e colaborações. Cada instância de *CRC Body* representa uma dessas responsabilidades, neste caso um cenário, e as respectivas classes colaborantes que a implementam.

4.3.2.3 *CRCManager*

A *CRCManager* é a classe que trata de criar cada cartão CRC, com todos os seus elementos, e de gerar o conjunto dos cartões sob o formato de um ficheiro

HTML. Esta classe disponibiliza três métodos públicos para o efeito: *produceCRCCards()* que gera a lista dos cartões, e *printCRCCards()* e *printCRCCards(String nfileName)*, que permitem escrever a lista no ficheiro. A sequência simplificada das acções executadas está representada na Figura 54, por um diagrama de sequência da UML.

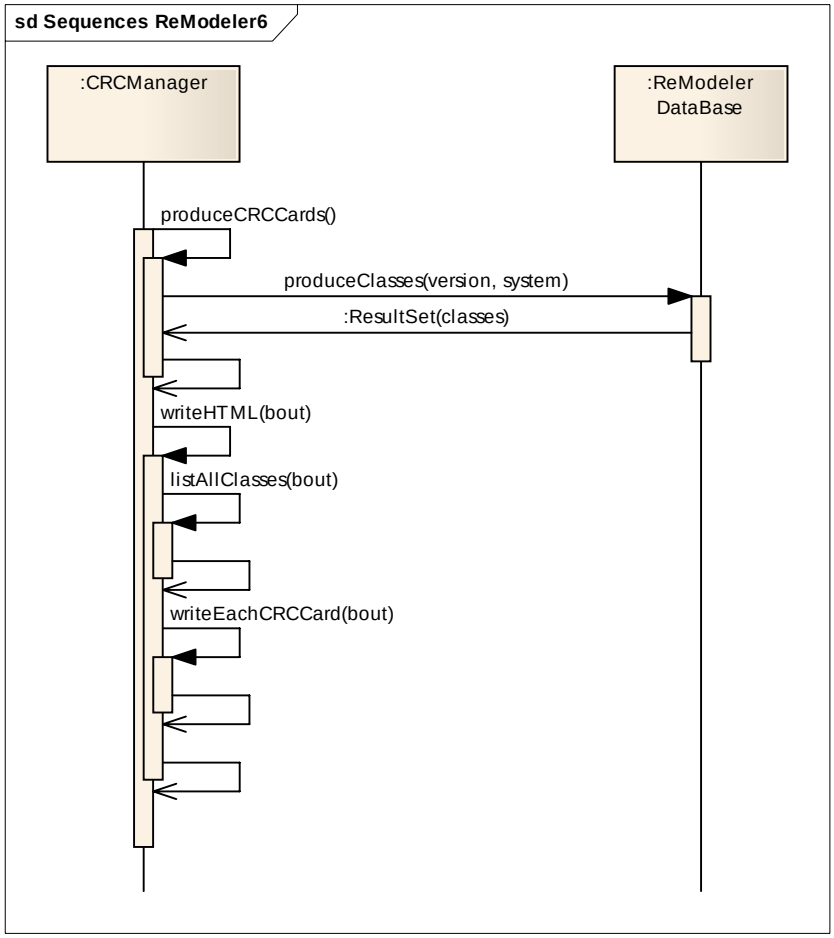


Figura 54. Diagrama de sequência para a produção de *Extended CRC Cards*.

4.3.3 Pacote *CRUD*

É no pacote CRUD que são criadas todas as matrizes CRUD para o sistema *ReModeler*. Este pacote está organizado segundo uma hierarquia de classes. A primeira classe é a classe *CRUDManager*, de onde herdam as classes *CRUDSimple* e *CRUDComplete*. Desta ultima vão herdar as classes *CRUDCounts* e *CRUDComplex*. A estrutura deste pacote pode ser vista no diagrama de classes da Figura 55. As

classes que pertencem a outros pacotes têm um papel auxiliar na execução das funcionalidades por este implementadas.

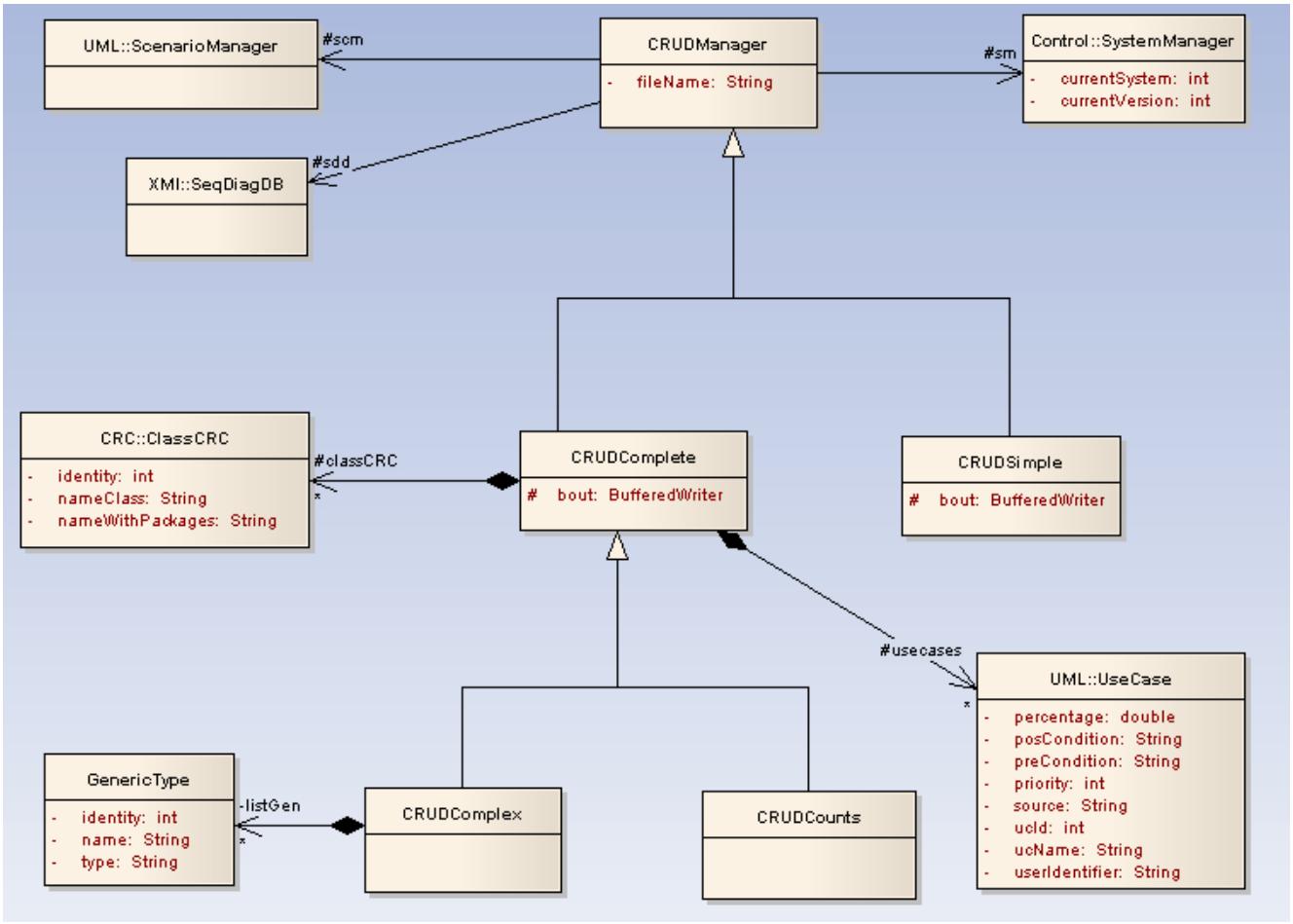


Figura 55. Diagrama de classes para o pacote CRUD.

4.3.3.1 *CRUDManager*

A classe *CRUDManager* vai iniciar a construção de qualquer uma das matrizes que são geradas para o sistema: matriz simples, matriz completa, matriz complexa e matriz com informação estatística. Para além disso ela disponibiliza um conjunto de métodos que são comuns à geração de qualquer das matrizes, como os métodos relacionados com o formato *HTML* (*writeInitialHTML (BufferedWriter), writeFinalHTML (BufferedWriter)*), métodos que identificam as operações invocadas por uma determinada classe para um determinado cenário (*produceOperation (int classid, int scenarioid)*) e ainda métodos que preparam os elementos das matrizes para serem escritos.

4.3.3.2 *CRUDSimple*

CRUDSimple é a classe que é responsável por gerar a matriz simples do sistema. Esta matriz apenas mostra os dados relativos a uma captura de um determinado cenário, apresentando também apenas as classes que estiveram directamente envolvidas na mesma. Para tal, a classe disponibiliza métodos que vão ler da base de dados as classes envolvidas na execução do cenário (*produceClassesListSimple (int scenarioId)*) e depois cruza essa informação com a obtida da utilização do método da classe *CRUDManager*, *produceOperation (int classid, int scenarioid),* que identifica as operações que foram invocadas. A informação é depois traduzida para um ficheiro em formato *HTML*. A Figura 56 mostra o diagrama de sequência representativo da geração da matriz simples. A comunicação com a base de dados é mostrada de uma forma simplificada, para não introduzir demasiada complexidade no diagrama.

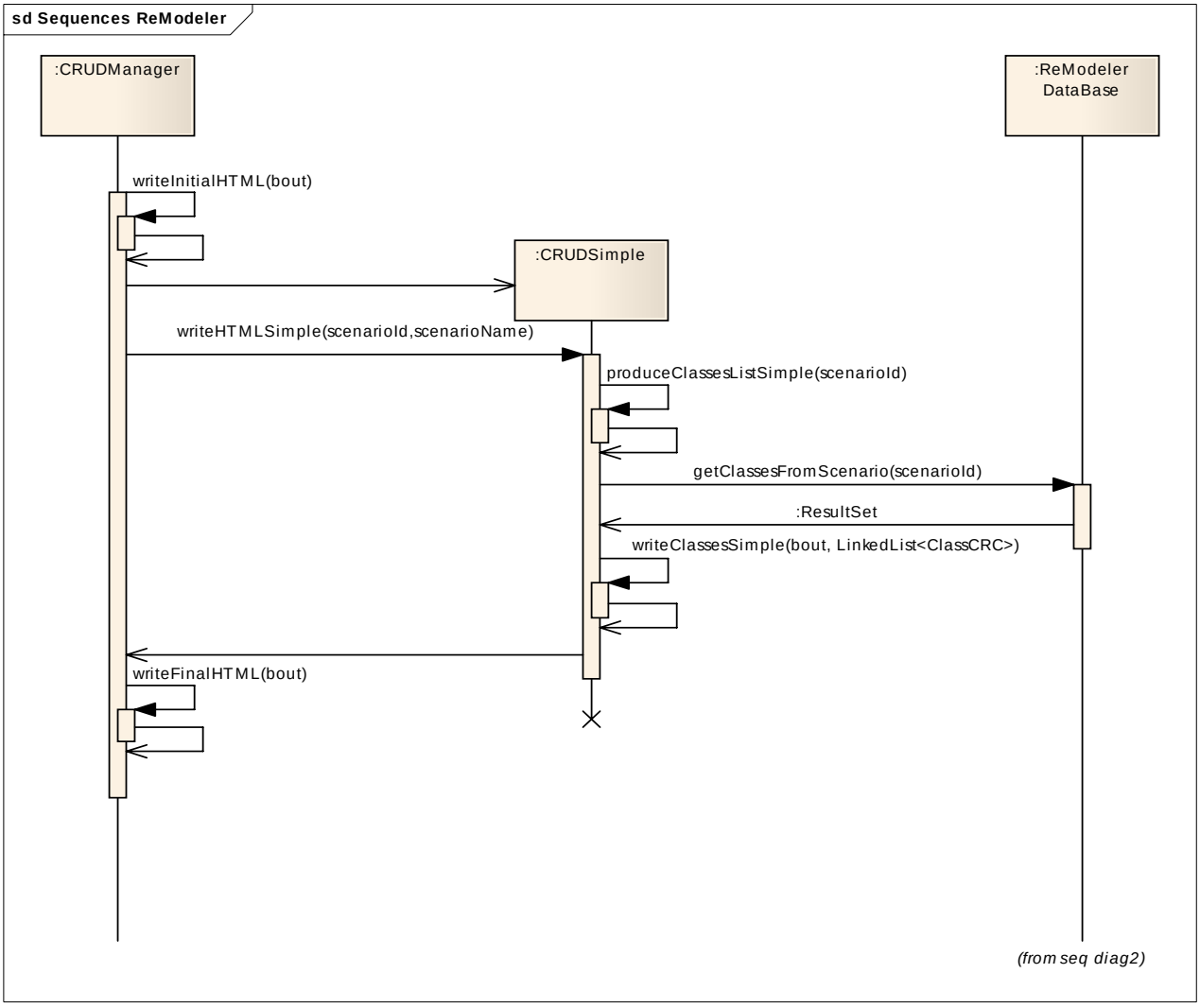


Figura 56. Diagrama de sequência para a criação de matriz simples.

4.3.3.3 *CRUDComplete*

A matriz completa do sistema é a matriz que mostra todos os cenários do sistema, agrupados por casos de utilização, independentemente de terem sido ou não capturados, em análise com todas as classes do sistema, importadas exteriormente através de um diagrama de classes.

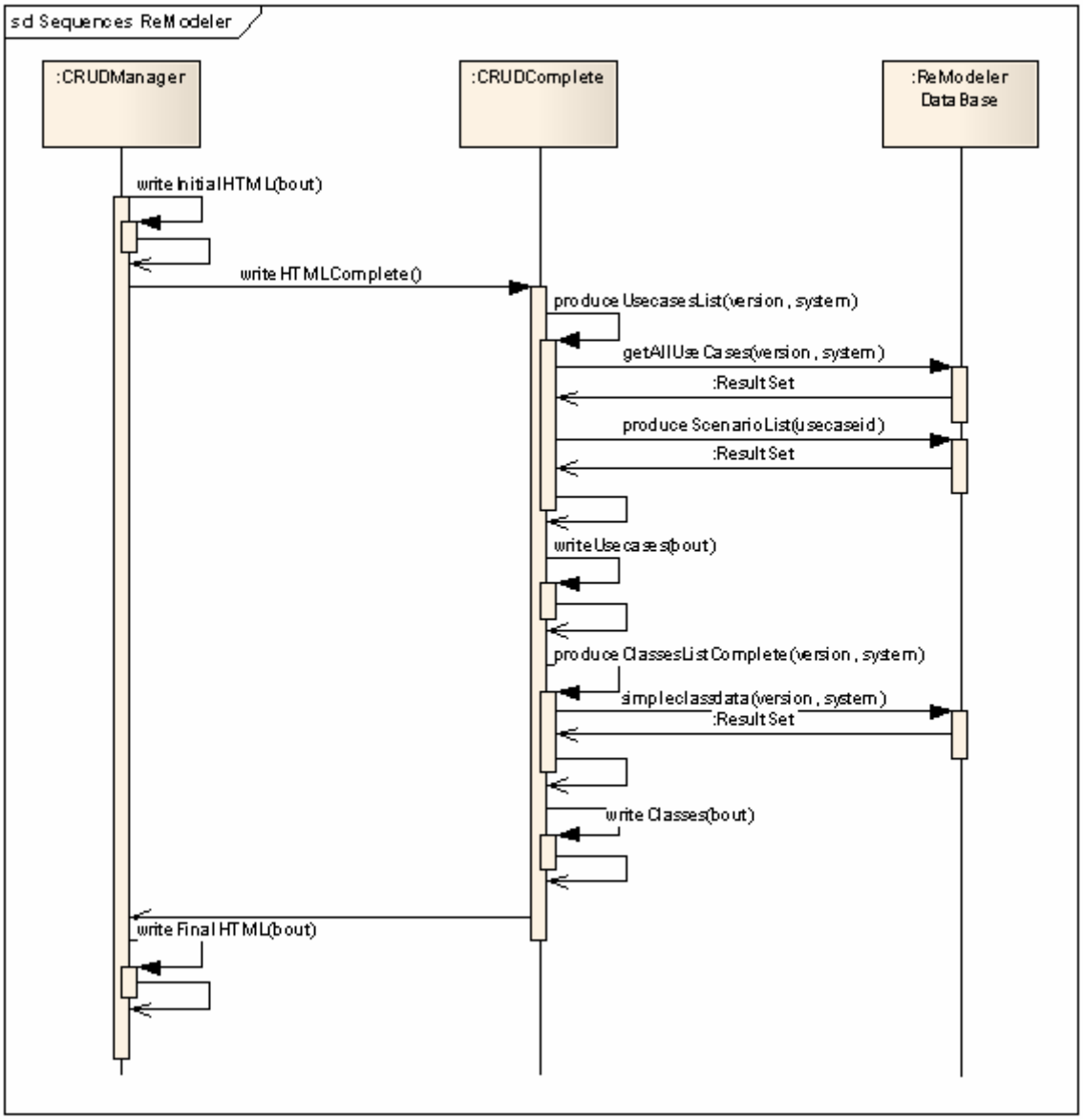


Figura 57. Diagrama de sequência para a criação de matriz completa.

A classe *CRUDComplete* é a classe que está incumbida de gerar e exportar para *HTML*, as matrizes completas. A produção destas matrizes é semelhante à das matrizes simples, embora com mais elementos. Inicia-se com a leitura da base de dados dos casos de utilização (*produceUsecasesList(int numVersion, int idSys)*) e dos respectivos cenários para o sistema *(produceScenarioList(int ucId)*). Depois são lidas

todas as classes do sistema (*produceClassesListComplete(int numVersion, int idSys)*) e é de novo invocado o método *produceOperation (int classid, int scenarioid)* da classe *CRUDManager*.

4.3.3.4 *GenericType*

Esta classe representa um tipo único para incluir nas matrizes, particularmente na matriz complexa. Uma vez que esta matriz pode ter diferentes graus de granularidade, a classe *GenericType* funciona como elemento comum e único a ser utilizado, podendo representar um pacote, classe ou método.

4.3.3.5 *CRUDComplex*

A matriz complexa é a matriz gerada pela classe *CRUDComplex*, no *ReModeler*, em que a granularidade dos seus elementos pode variar. No lugar das classes, das matrizes descritas anteriormente, esta matriz permite ter níveis de abstracção diferentes, como pacotes, classes ou métodos, dependendo do que o analista deseja analisar. Inicialmente é disponibilizada uma árvore idêntica à da filtragem, para que o utilizador escolha os elementos e respectiva granularidade para os quais quer gerar a matriz. Isto origina a produção de uma lista de elementos *GenericType* que é traduzida depois dentro da classe, através do método *writeComplexClasses()*. Para cada tipo de elemento são invocados métodos internos, idênticos ao *produceOperation (int classid, int scenarioid)* da classe *CRUDManager*, que determinam as invocações que foram realizadas ao nível dos pacotes e dos métodos.

4.3.3.6 *CRUDCounts*

Por último, é a classe *CRUDCounts* que está encarregue de gerar as matrizes com informação estatística. Estas matrizes são muito idênticas às matrizes completas, mas nas suas extremidades são adicionas uma linha e uma coluna, onde se coloca informação sobre o número de operações que ocorreram. Mais precisamente, em cada célula da linha adicionada é apresentado o somatório das invocações de métodos que representam operações dos tipos “C”,”R”,”U” e ”D”, para cada cenário. Do mesmo modo, em cada célula da coluna adicionada é colocado o somatório das invocações de métodos que representam operações dos tipos “C”,”R”,”U” e ”D”,

para cada classe. Esta classe herda os métodos da classe *CRUDComplete* e adiciona novos para permitir contar e escrever o número de invocações, *writeScenarioCount(BufferedWriter bout), getScenarioCount(int scenarioID), writeClassesCounts(BufferedWriter bout), getClassCount(int identity),* entre outros.

4.3.4 Pacote *Capture*

A base de todo o funcionamento do sistema *ReModeler* está na captura do sistema em análise. Todos os artefactos descritos nesta dissertação são gerados graças ao tratamento que é feito aos dados obtidos das capturas internas da execução dos cenários do sistema. O pacote que implementa a captura interna do sistema é o pacote *Capture*. Para realizar uma captura, o sistema em análise deve ser entrelaçado com o *ReModeler* e devem ser os dois compilados juntos. Isto é possível através das facilidades de *weaving* da tecnologia dos aspectos. A Figura 58 mostra o diagrama de classes deste pacote, constituído por três classes Java, *ContextClass, MessageManager e CaptureManager*, e um aspecto, *CaptureAspect*. Mais uma vez, as classes presentes no diagrama que não pertencem ao pacote, têm funções auxiliares para a execução das funcionalidades.

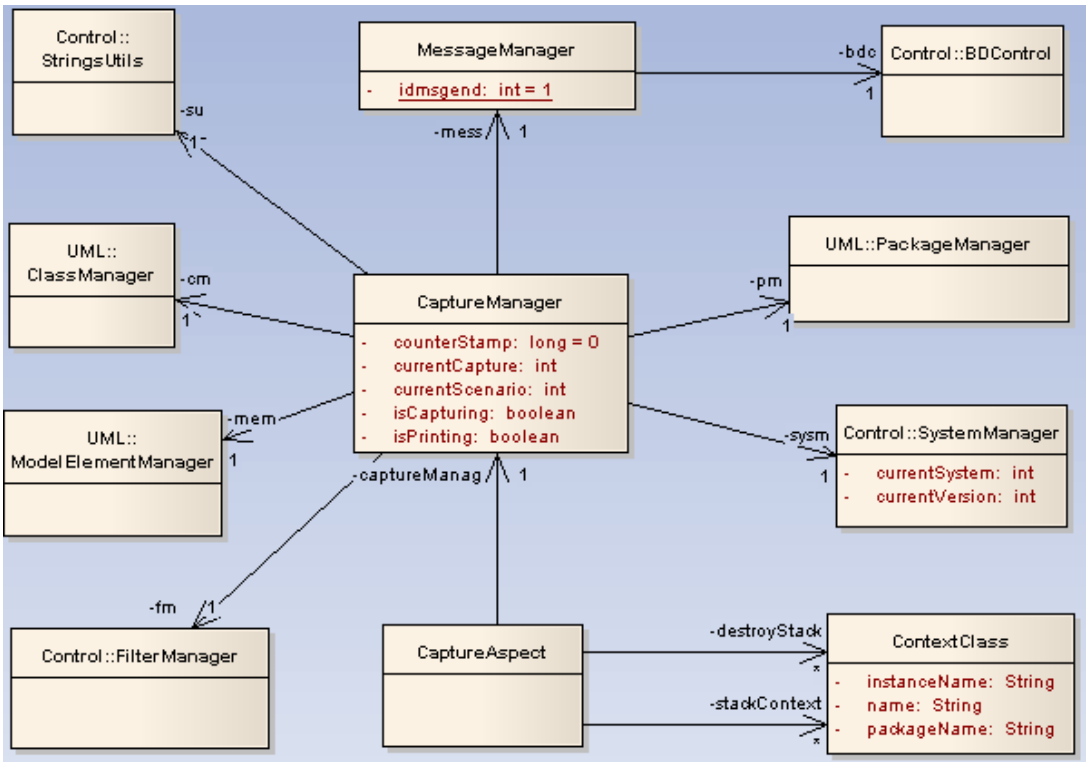


Figura 58. Diagrama de Classes para o pacote Capture.

4.3.4.1 *CaptureAspect*

Quando se quer fazer uma captura de execução de um sistema, existem alguns factores que têm de ser estudados, como por exemplo o modo de instrumentação do código em análise. A estratégia de instrumentação tem de ser guiada de modo a permitir recolher as informações necessárias, em tempo de execução, minimizando o máximo possível o impacto na execução (*overhead)* normalmente associado a esta actividade. Na implementação do *ReModeler* foi usada uma estratégia de instrumentação baseada na programação orientada a aspectos (*Aspect-Oriented Programming* - AOP) [Wikipedia, 2008], de maneira a tornar essa instrumentação, o menos intrusiva possível (ver anexo A).

A classe *CaptureAspect* é um aspecto constituído por três *advices* e respectivos *pointcuts*, como mostra a Figura 59.

O primeiro e o segundo *advice* foram implementados para serem executados em vez (*around*) da execução de qualquer método, de qualquer classe ou pacote. A diferença entre ambos é que o primeiro deve apanhar as execuções de métodos que estejam declarados como estáticos *execution(static * *..*.*(..))*, sendo que o * significa qualquer. Cada vez que este *advice* é activado, são criadas mensagens de *call*, *destroy* e *exit*, como mostra a Figura 59, que são armazenadas na base de dados. Para além disso é utilizado o método *proceed()* para que o código Java do sistema original seja executado.

O terceiro *advice*, também da Figura 59, é responsável por capturar as criações de objectos que ocorrem durante a execução. Ele difere dos outros dois pois especifica que o método invocado tem de ser um método *new*, *execution(*..*.new(..))*, ou seja, uma criação. Sempre que existe uma criação o *advice* vai originar uma mensagem de *create*, como a que está na Figura 60, que é armazenada na base de dados.

```
//Captures static calls
pointcut staticSequence():
      execution(static * *..*.*(..)) && erroneousWay();

Object around(): staticSequence(){
                        (…)
      Object[] returnArg = {proceed()};
                        (…)
      return returnArg[0];
}
//Captures non static calls
Object around(Object self): execution(* *..*.*(..))
      && erroneousWay()&& !execution(void Runnable+.run(..))
      && this(self){
                        (…)
      Object[] returnArg = {proceed(self)};
                        (…)
      return returnArg[0];
}
//Captures object creations
Object around(Object self): execution(*..*.new(..))
      && erroneousWay()&& this(self){
                        (…)
      proceed(self);
                        (…)
      return null;
}
```

Figura 59. Estrutura do aspecto do *ReModeler*.

```
insertCallMessage("Call", ContextClass, ContextClass,
thisJoinPointStaticPart.getSignature().getName(),thisJoinPoint.getArgs(), true);

insertDestroyMessage("Destroy", ContextClass, returnArg,
stackContext.getLast());

insertExitMessage("Exit", ContextClass, ContextClass, returnArg,
thisJoinPointStaticPart.getSignature().getName(), thisJoinPoint.getArgs());

insertCreateMessage("Create", stackContext.getLast(),ContextClass);
```

Figura 60. Métodos que criam as mensagens.

4.3.4.2 *ContextClass*

A classe *ContextClass* é a classe que representa um invocador ou receptor de uma mensagem. Quando uma mensagem é invocada, esta pode envolver quer uma classe (método estático), quer uma instância de uma classe.

4.3.4.3 *CaptureManager*

Esta classe *CaptureManager* inicia e gere cada captura. Sempre que é pedida uma nova captura, esta classe activa uma variável *isCapturing* que permite ordenar a recolha de informação e inicializar na base de dados os elementos necessários. Para além disso, esta classe disponibiliza um conjunto de métodos que são depois invocados no aspecto *CaptureAspect* para criação de mensagens, classes, instâncias, etc (ver Figura 60). Um problema interessante, que é necessário resolver, neste tipo de aplicações é a identificação única de uma instância ou classe interveniente. Nesta implementação, a maneira encontrada para resolver este problema foi criar um método, *createUniqueName*, que dado o objecto que se pretende identificar, gera um identificador único através do *hashcode* do objecto, no método *createUniqueID* da classe *StringUtils*, e o conjuga com nome da própria instância. Por sua vez, o nome da instância necessita de ser preparado para escrita, através do método *transformNames*, que vai substituir os caracteres especiais que vêm muitas vezes incluídos na *String* do nome, como '@', por '.'. O código referente ao método *createUniqueName* é mostrado na Figura 61.

```
/*
* Cria o nome de uma instancia, conjugando
* o nome da classe com o seu id único.
* su é instância de StringsUtils do pacote Control.
*/
public String createUniqueName(Object obj){
  String uniqueID = su.createUniqueID(obj);
  String classname = su.transformNames(obj.getClass().getName());
  return (classname+"."+uniqueID);
}
```

Figura 61. Método que cria um nome único para uma instância.

4.3.4.4 *MessageManager*

A classe *MessageManager* é a classe incumbida de invocar as funções necessárias à base de dados, para que fiquem armazenadas de modo persistente, as mensagens recolhidas nas capturas. Um exemplo de um dos métodos existentes nesta classe pode ser visto na Figura 62.

```
public long insertMessage(String type, int source, int dest,
            long st, int cap) {
  try {
      Statement stmt = bdc.initDriverConnection();
      stmt.executeQuery("SELECT \"setNewMessage\"('"+st+ "','"+type+
              "','"+cap+"','"+source+"','"+dest+"');");
      bdc.closeDriverConnection(stmt);
  } catch (Exception e) {    e.printStackTrace();}
  return st;
}
```

Figura 62. Implementação de um método da classe MessageManager, que armazena mensagens na base de dados.

4.3.5 Pacote *XMI*

Durante uma captura de um cenário do sistema em análise, os dados recolhidos são guardados na base de dados do *ReModeler*. A geração de qualquer dos artefactos propostos só é conseguida pela leitura e tratamento desses mesmos dados. Isto é ainda mais importante quando o artefacto a gerar é um diagrama de sequência temporizado. Cada diagrama de sequência gerado vai representar, de modo diagramático, as mensagens que foram trocadas entre os objectos do sistema, aquando da execução. A leitura e o tratamento dos dados necessários, assim como a sua posterior exportação para ficheiros XMI, são responsabilidades do pacote XMI. É também aqui que é gerado o conjunto de mensagens que constituem a linguagem interna do *ReModeler* para ser fornecida ao componente *VisualScenarioGenerator*, que depois lhe vai permitir animar os diagramas de sequência, descrito em [Gouveia, 2008]. Este pacote é constituído por seis classes, como mostra a Figura 63, que se relacionam com as restantes para executar as funcionalidades descritas.

4.3.5.1 *XmiUtils* e *EALocation*

Actualmente, as ferramentas de UML ainda não suportam, na sua totalidade, a importação e exportação de ficheiros XMI. De entre as ferramentas testadas podem ser destacadas o *MagicDraw* UML [MagicDraw, 2008], o *Objecteering* [Objecteering, 2008], o *Altova UModel* [Altova, 2008] e o *Enterprise Architecture* [SparxSystems, 2008]. Em algumas destas, o comportamento apresentado é tão insuficiente que nem sequer conseguem importar o próprio ficheiro por elas gerado. As que se comportam de um modo mais razoável, conseguem importar e exportar os diagramas, mas exigem um conjunto de extensões aos ficheiros XMI originais tal, que o seu tamanho chega a quadruplicar. Enquanto estas ferramentas não melhoram

o seu modo de interoperabilidade, foi escolhido o *Enterprise Architecture* que pareceu apresentar os melhores resultados. Deste modo, os diagramas XMI gerados foram adaptados às extensões exigidas por essa ferramenta. As duas classes, *XmiUtils* e *EALocation* foram implementadas para permitirem a gestão dos elementos necessários a essa extensão, através do fornecimento de um conjunto de métodos de apoio à geração.

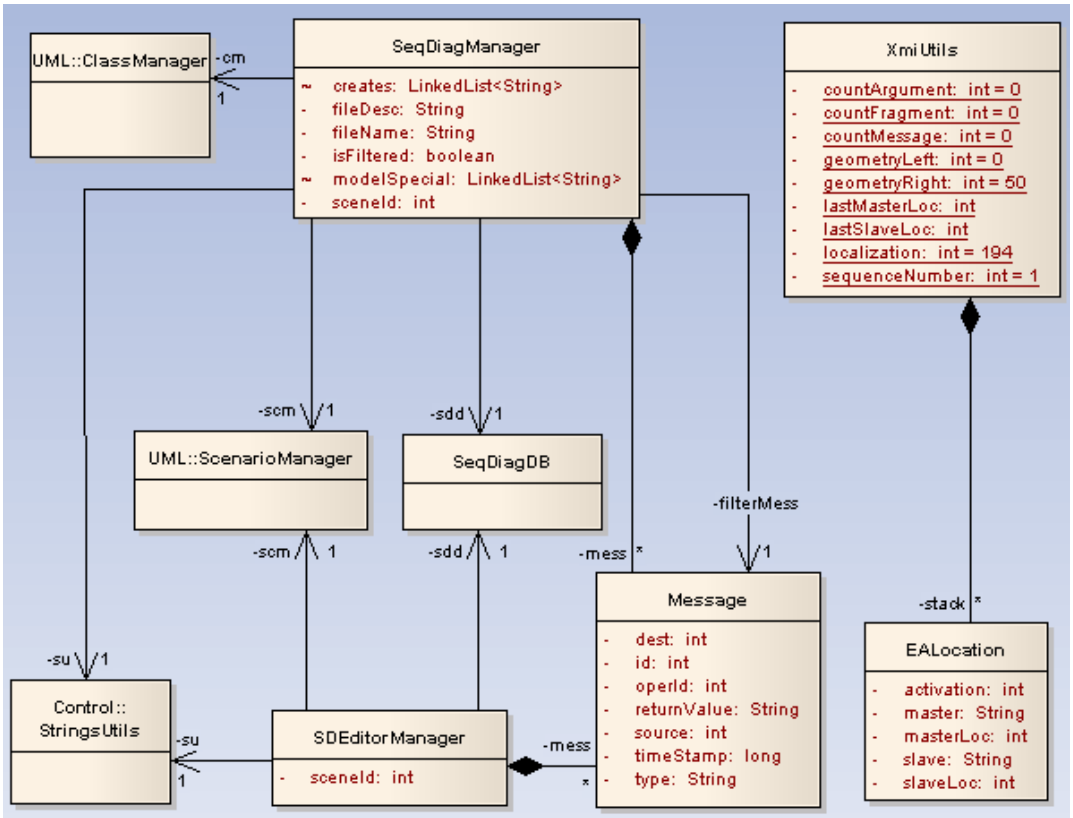


Figura 63. Diagrama de classes do pacote XMI.

4.3.5.2 *Message*

A classe *Message* representa uma mensagem do diagrama de sequência. Aqui estão representadas os campos que caracterizam uma mensagem, como o seu tipo, o seu identificador, os elementos de origem e destino e a sua marca temporal. Uma marca temporal corresponde ao instante de tempo em que a mensagem foi invocada aquando de uma captura.

4.3.5.3 *SeqDiagDB*

Como já foi referido anteriormente, a geração de cada diagrama de sequência é conseguida pela leitura e análise dos dados que estão armazenados na base de dados. A classe *SeqDiagDB* é responsável por essa leitura. É ela que comunica com a

base de dados, invocando as funções necessárias para recolher os dados para tratamento. Alguns dos métodos que a classe disponibiliza estão apresentados na Figura 64.

public String getMessageReturnValue(Message message): devolve o valor de retorno de uma determinada mensagem.
public String getMessageInstance(int msgEnd): devolve a instância onde a mensagem foi invocada.
private Message getMessageData(String result): extrai os dados de uma mensagem que são depois carregados numa instância da classe *Message*.

Figura 64. Alguns métodos implementados pela classe *SeqDiagDB*.

4.3.5.4 *SeqDiagManager*

A classe *SeqDiagManager* é a classe responsável pela gestão da produção e geração dos diagramas de sequência, que vão ser exportados para ficheiros XMI. Ela disponibiliza métodos de escrita das várias partes que constituem um ficheiro XMI, que estão descritas esquematicamente na Figura 65. Para além disso, implementa os métodos que tratam dos dados recolhidos pela classe *SeqDiagDB*, para estarem no formato específico do XMI.

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1" xmlns:uml="http://schema.omg.org/spec/UML/2.1" ... >
        <xmi:Documentation .../>
        <uml:Model xmi:type="uml:Model" name="..." visibility="...">
                <packagedElement xmi:type="uml:Package|uml:Actor|uml:UseCase|uml:Class|..."
xmi:id="..." name="..." visibility="...">      (...)       </packagedElement>
        </uml:Model>
        <xmi:Extension extender="..." extenderID="...">
                <elements>
                        <element xmi:idref="..." xmi:type="..." name="..." scope="..">
                                <model ..."/><properties .../><style .../>…
                        </element> (...)
                </elements>
                <connectors>
                        <connector xmi:idref="…">
                                <source xmi:idref="..."> (...) </source>
                                <target xmi:idref="..."> (...) </target>
                                (...)
                        </connector>
                </connectors>
                <diagrams>
                        <diagram xmi:id="…"> (...)
                                <elements>
                                        <element geometry="..." subject="..." seqno="..." style="..."/> (...)
                                </elements>
                        </diagram>
                </diagrams>
        </xmi:Extension>
</xmi:XMI>
```

Figura 65. Sintaxe simplificada de um ficheiro XMI.

Ao se observar a Figura 65 é possível verificar as limitações descritas anteriormente, impostas pelas ferramentas UML. Num ficheiro XMI, apenas as partes delimitadas por *<uml:Model> …</uml:Model>* deveriam ser necessárias para descrever os elementos a representar em cada diagrama. No entanto, torna-se também necessário adicionar a informação compreendida entre as *tags* *<xmi:Extension>* e *</xmi:Extension>*, que corresponde à exigida pela ferramenta. Como se pode ver, pela figura, esta última parte representa mais de ¾ do ficheiro em causa.

4.3.5.5 *SDEditorManager*

Na classe *SDEditorManager* são também gerados diagramas de sequência, mas com um propósito diferente do da classe *SeqDiagManager*. Também aqui, os dados de cada captura são lidos da base de dados, através da classe *SeqDiagDB*, mas o seu tratamento vai ser diferente. Os diagramas aqui gerados estão num formato interno e proprietário do *ReModeler* que é passado ao componente *VisualScenarioGenerator*, para ser posteriormente animado [Gouveia, 2008]. Esta classe disponibiliza então dois métodos principais, *getCreations(int capId)* e *getMessages()*, em que o primeiro corresponde à criação dos elementos do diagrama (*lifelines*) e o segundo corresponde a todas as mensagens suportadas pela linguagem, os *creates*, os *calls* e os *destroys*.

4.3.6 Pacote *UML*

O pacote UML não tem uma funcionalidade definida. Este pacote agrupa um conjunto de classes que representam os conceitos de base, quer do *ReModeler*, quer da linguagem UML (ver Figura 66). Para além da classe que representa cada elemento, existe também uma classe que representa o respectivo gestor. Um gestor tem como função tratar de todas as acções relacionadas com a classe a que se refere. Isto é realizado através da invocação das funções presentes na base de dados, que permitem fazer inserções, leituras, actualizações e remoções. Por exemplo, a classe Actor, que representa um elemento actor de um diagrama de casos de utilização UML, tem associada a classe *ActorManager*, que figura o seu gestor. Neste caso, a classe *ActorManager* disponibiliza um conjunto de métodos, sucintamente descritos na Figura 67, que permitem inserir, remover ou pesquisar um actor, da base de dados

do *ReModeler*. De um modo semelhante ao que acontece com o actor, também os outros elementos constituintes do pacote, como o caso de utilização (*UseCase*), o cenário (*Scenario*), o pacote (*Package*), etc, têm associado o seu respectivo gestor.

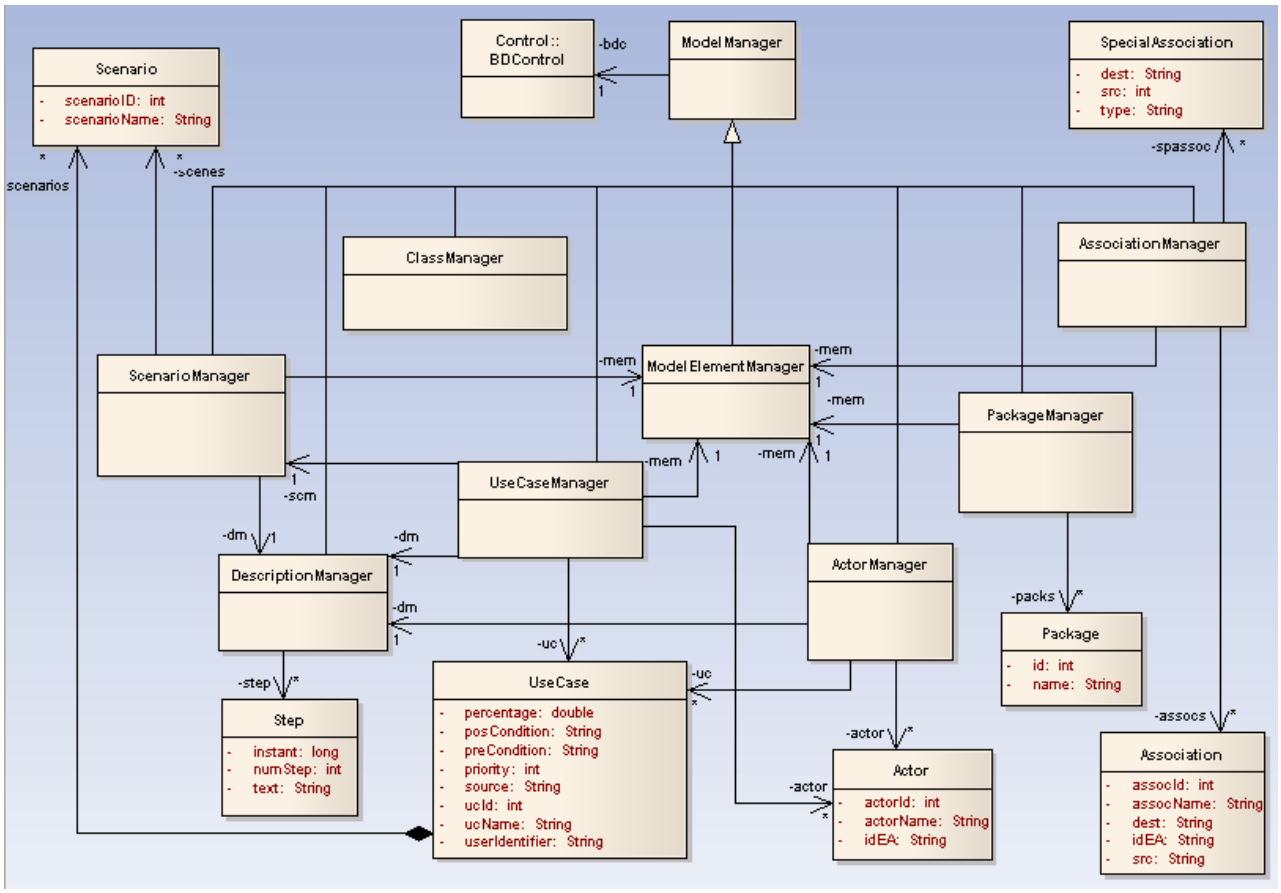


Figura 66. Diagrama de classes do pacote UML.

```
public boolean insertNewActor(String idElem, String actorName, int
sysId, int numVersion);

public void deleteActor(String idElem, String actorName, int sysId,
int numVersion);

public boolean updateActorName(String oldName, String newName,
String oldIdElem, String newIdElem, int sysId, int numVersion);

public int getActorid(String idElem, String actorName, int sysId,
int numVersion);
                                    (…)
```

Figura 67. Alguns dos métodos disponibilizados pela classe *ActorManager*.

4.3.7 Pacote *SAXImport*

Artefactos como os diagramas coloridos ou as matrizes de CRUD necessitam de ter informação adicional à obtida dinamicamente. Essa informação é obtida, no *ReModeler*, pela importação de diagramas estáticos, como diagramas de classes ou diagramas de casos de utilização.

O pacote *SaxImport* é o pacote responsável por importar os diagramas e inserir os seus dados na base de dados do *ReModeler*, para posterior utilização. Este pacote é constituído por oito classes, representadas no diagrama da Figura 68.

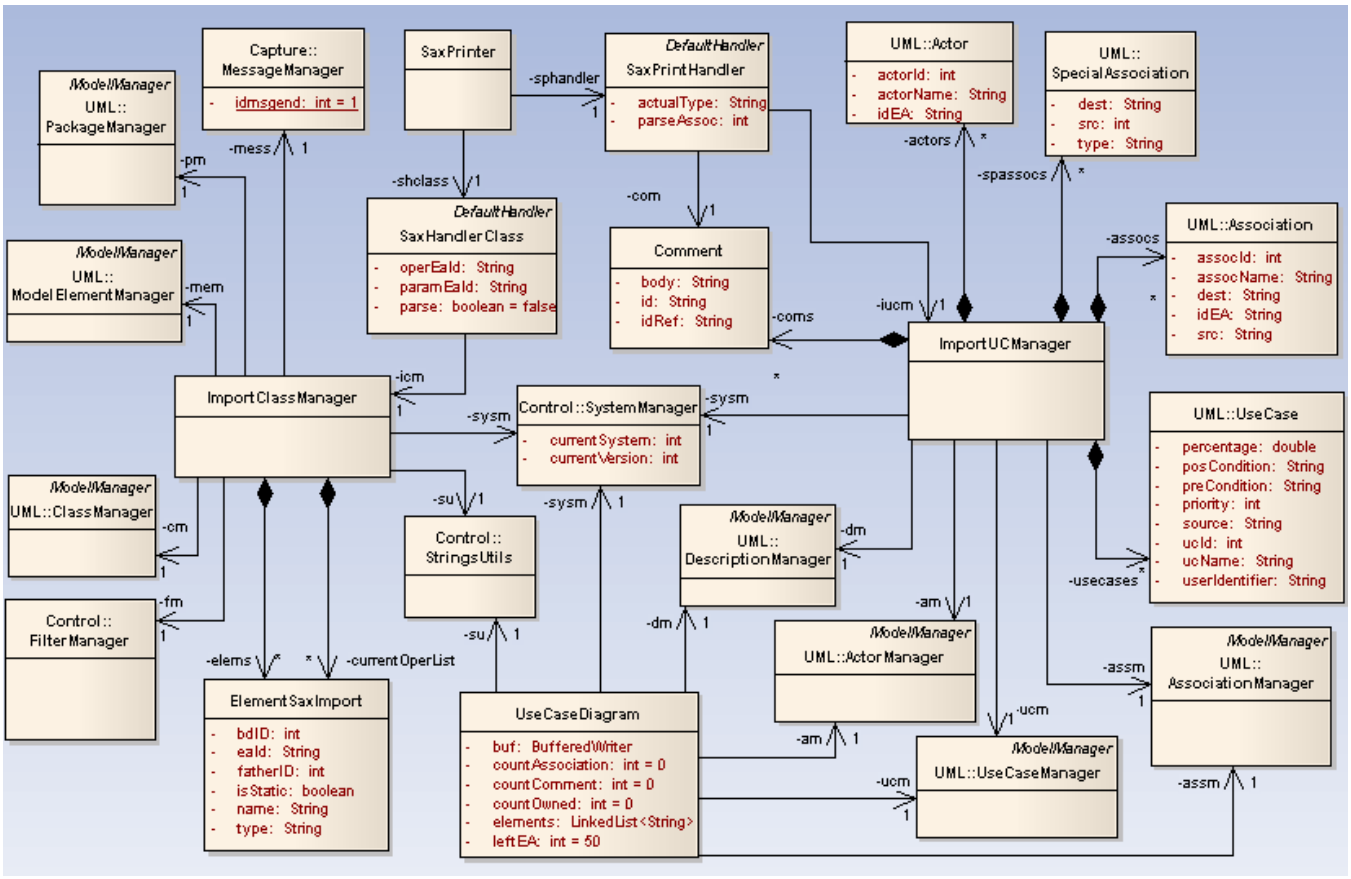


Figura 68. Diagrama de classes do pacote *SaxImport*.

4.3.7.1 *SaxPrinter*

A importação dos diagramas UML em formato XMI é feita com recurso a uma API, bastante conhecida, para interpretar ficheiros XML na linguagem Java, chamada SAX [Network World, 2008] (ver anexo A). A classe *SaxPrinter* é a classe que vai criar o objecto *Sax Parser*, passando-lhe o ficheiro e o *handler*, que por sua

vez vai tratar os elementos lidos do ficheiro. Na Figura 69 é possível ver o código fonte de um dos métodos existentes nesta classe.

```
public static void importerUC(String file, SystemManager sysm) throws
Exception {
	SAXParserFactory factory = SAXParserFactory.newInstance ();
	XMLReader xmlReader = factory.newSAXParser().getXMLReader();
	SaxPrintHandler handler = new SaxPrintHandler(sysm);
	xmlReader.setContentHandler(handler);
	xmlReader.setErrorHandler(handler);
	FileReader reader = new FileReader(file);
	xmlReader.parse(new InputSource(reader));
}
```

Figura 69. Implementação de um método da classe SaxPrinter.

4.3.7.2 *ElementSaxImport*

Qualquer elemento que é lido do documento XMI é representado por esta classe. Ela contém os dados comuns e necessários à identificação de cada elemento, como o nome, o identificador no documento, o tipo, os parâmetros, entre outros. A maioria desses dados é lida directamente do ficheiro, através da sintaxe definida pelo XMI, como mostrado na Figura 70.

```
<packagedElement xmi:type="uml:Actor" xmi:id="EAID_DC3B373D_07F7_47a3_9398_97AD6734D434"
name="User" visibility="public"/>

<packagedElement xmi:type="uml:Association" xmi:id="EAID_019DF3AA_87E2_4a8c_881A_9C69968B0E6E"
visibility="public">

<packagedElement xmi:type="uml:UseCase" xmi:id="EAID_75289E79_C1D8_4e35_A403_A925A727C3F0"
name="Change ball radius" visibility="public"/>

<packagedElement xmi:type="uml:Class" xmi:id="EAID_BD00A6D5_B1B4_4779_8149_B9551CBF363D"
name="PongFrame" visibility="public">
```

Figura 70. Exemplos da sintaxe do XMI para elementos dos diagramas.

4.3.7.3 *Comment*

Esta classe representa os comentários que podem vir especificados nos diagramas XMI. Pela Figura 71 é possível verificar que a sintaxe deste elemento é um pouco diferente dos anteriores, daí ter existido a necessidade de criar uma classe separada.

```
<ownedComment xmi:type="uml:Comment" xmi:id="comment01" annotatedElement="thecustomprofile">
                <body> Version:1.0</body>
</ownedComment>
```

Figura 71. Exemplos da sintaxe do XMI para comentário.

4.3.7.4 *SaxPrintHandler* e *SaxHandlerClass*

Cada uma destas classes representa um *document handler*, que recebe os *callbacks* para os eventos SAX. Sempre que um evento ocorre, ele é passado ao método que foi definido para o tratar. Para que tal aconteça, as classes derivam de *org.xml.sax.helpers.DefaultHandler*. A classe *SaxPrintHandler* trata da importação dos diagramas de casos de utilização e disponibiliza para tal três métodos (Figura 72, parte A). O primeiro método regista o início do ficheiro a interpretar, o segundo identifica o fim desse ficheiro e o terceiro trata dos restantes elementos XML encontrados ao longo do ficheiro. De modo identico, a classe *SaxHandlerClass* trata da importação dos diagramas de classes. Para além dos ter os mesmos três métodos, definidos na classe anterior, ela disponibiliza ainda um quarto método (Figura 72 parte B), para tratar da finalização de cada elemento encontrado no ficheiro.

```
A:
public void startDocument() {…}
public void endDocument() {…}
public void startElement(String uri, String name, String qName,
      Attributes atts) {…}

---

B:
public void endElement(String namespaceURI, String sName,
      String qName){…}
```

Figura 72. Métodos implementados nas classes *SaxPrinterHandler* e *SaxHandlerClass*.

4.3.7.5 *ImportUCManager* e *ImportClassManager*

A importação de diagramas UML no sistema vai originar a inserção de novos elementos na base de dados. No caso da importação de um diagrama de classes do sistema em análise, os elementos descritos no ficheiro XMI, como classes, pacotes, etc, devem ser inseridos na base de dados nas tabelas correspondentes. No entanto, essa inserção não é imediata. Ela vai depender da verificação da existência prévia do elemento no sistema. Se esta verificação não for feita, a importação do mesmo diagrama *n* vezes, vai originar a replicação dos elementos *n* vezes na base de dados. Para além do mais, quando ocorre uma captura, existem elementos que são

introduzidos no sistema, que também podem entrar em conflito com os elementos importados. Por exemplo, uma classe *X* foi identificada na captura de uma execução e foi devidamente introduzida na base de dados, juntamente com dois métodos. No entanto, quando se faz a importação do diagrama de classes, a mesma classe é identificada com quatro métodos. É então feita a verificação de existência de cada um dos elementos no sistema e apenas os que ainda não constam, são inseridos.

A inserção e actualização na base de dados, dos elementos importados, está a cargo das duas classes *ImportUCManager* e *ImportClassManager*, que tratam dos elementos dos diagramas de casos de utilização e classes, respectivamente.

4.3.7.6 *UseCaseDiagram*

Para além da funcionalidade de importação de diagramas, também a exportação dos mesmos é possível, tal como descrito no capítulo 2. Existem duas formas possíveis de iniciar a documentação de um sistema no *ReModeler*: importar um diagrama de casos de utilização previamente construído numa ferramenta UML externa ou criar os casos de utilização directamente no editor do *ReModeler*. Em qualquer destas opções pode ser interessante exportar a informação presente no editor para um formato diagramático, ou seja, exportar a informação para um diagrama de casos de utilização em formato XMI. A classe *UseCaseDiagram* tem a responsabilidade de fazer essa exportação através do método *writeXmiUC()*. A implementação deste método, descrito sucintamente no diagrama de sequência da Figura 73, vai utilizar um conjunto de métodos privados auxiliares que agregam os elementos necessários à completude do diagrama. Inicialmente, é invocado o método *writeActors()*, que escreve no ficheiro, todos os actores do sistema, identifcados pela pesquisa na base de dados. Além disso, ele também invoca os métodos *writeSpecialAssActor(LinkedList<SpecialAssociation>, actorid)* e *writeComments (modelElementid)*, que escrevem todas as associações de generalizações e todos os coméntários, respectivamente, associados a cada actor. Depois é chamado o método *writeUseCases()* para, de modo similar ao dos actores, escrever no ficheiro todos os casos de utilização que estiverem na base de dados e respectivas associações, quer entre eles, quer com os outros elementos do diagrama, como actores, etc. Por fim, é invocado o método *writeDiagram()* que adiciona informação necessária à importação na ferramenta, descrita na secção 1.2.5 deste capítulo.

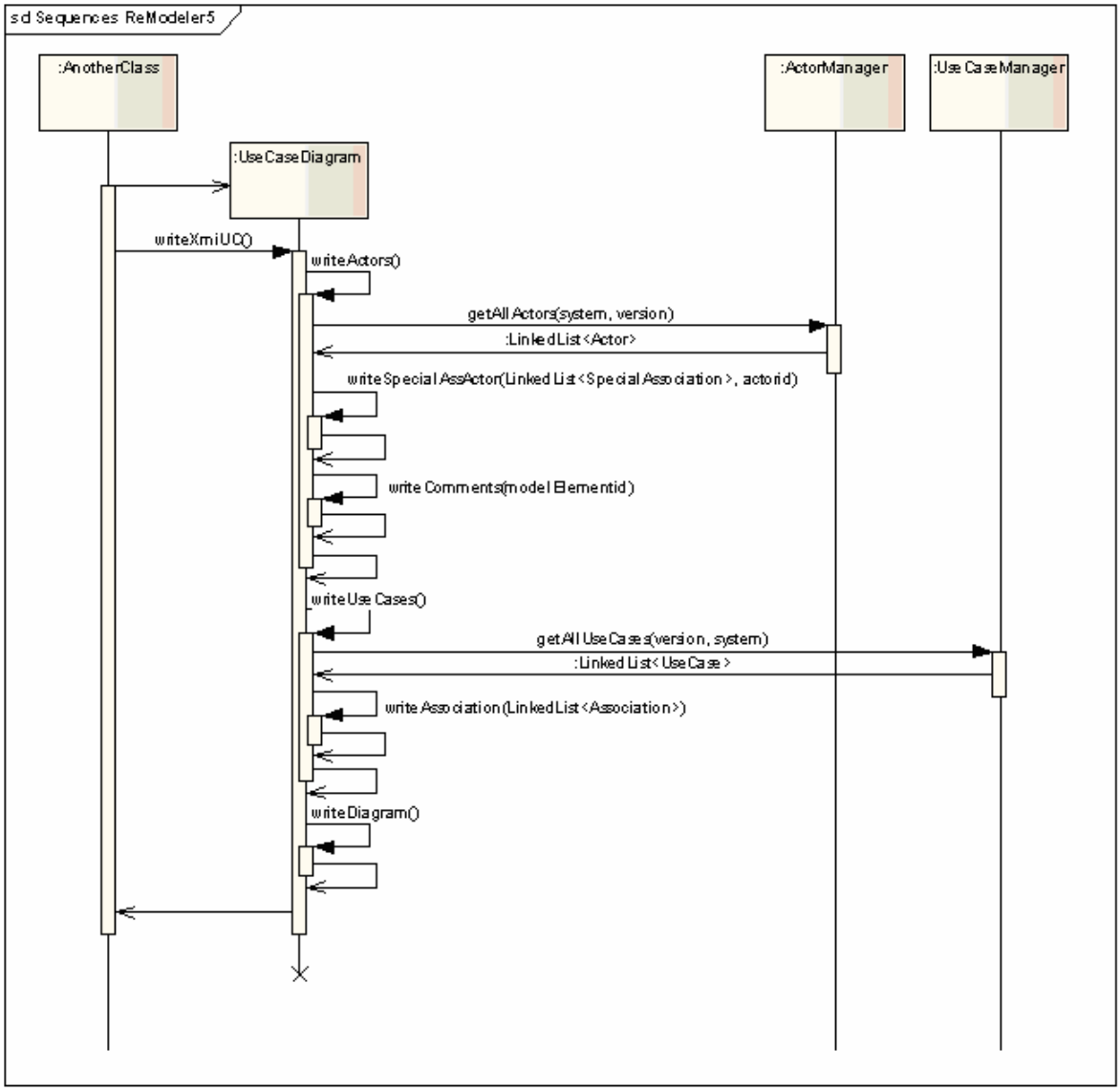


Figura 73. Diagrama de Sequência simplificado da execução da exportação do diagrama de casos de utilização.

4.3.8 Pacote *Color*

Dos artefactos que são produzidos, falta apenas falar dos diagramas coloridos. Recorda-se que o *ReModeler* tem a capacidade de gerar três tipos de diagramas coloridos, o diagrama de casos de utilização, o diagrama de classes e o diagrama de sequência descrito em [Gouveia, 2008]. A coloração dos elementos em cada um dos diagramas vai depender da percentagem de cobertura ou da intensidade de utilização respectiva. Actualmente, é possível ao utilizador escolher a palete de cores e o respectivo intervalo de percentagens que representa, para o caso da cobertura/intensidade de utilização das classes. No caso dos diagramas de casos de utilização, esses dados já estão definidos internamente à priori. O pacote *Color* é o

pacote que agrupa as classes que têm a responsabilidade de gerar os dois tipos de diagramas e de gerir os intervalos de cores. Ele é constituído por nove classes, representadas pelo diagrama de classes da Figura 74.

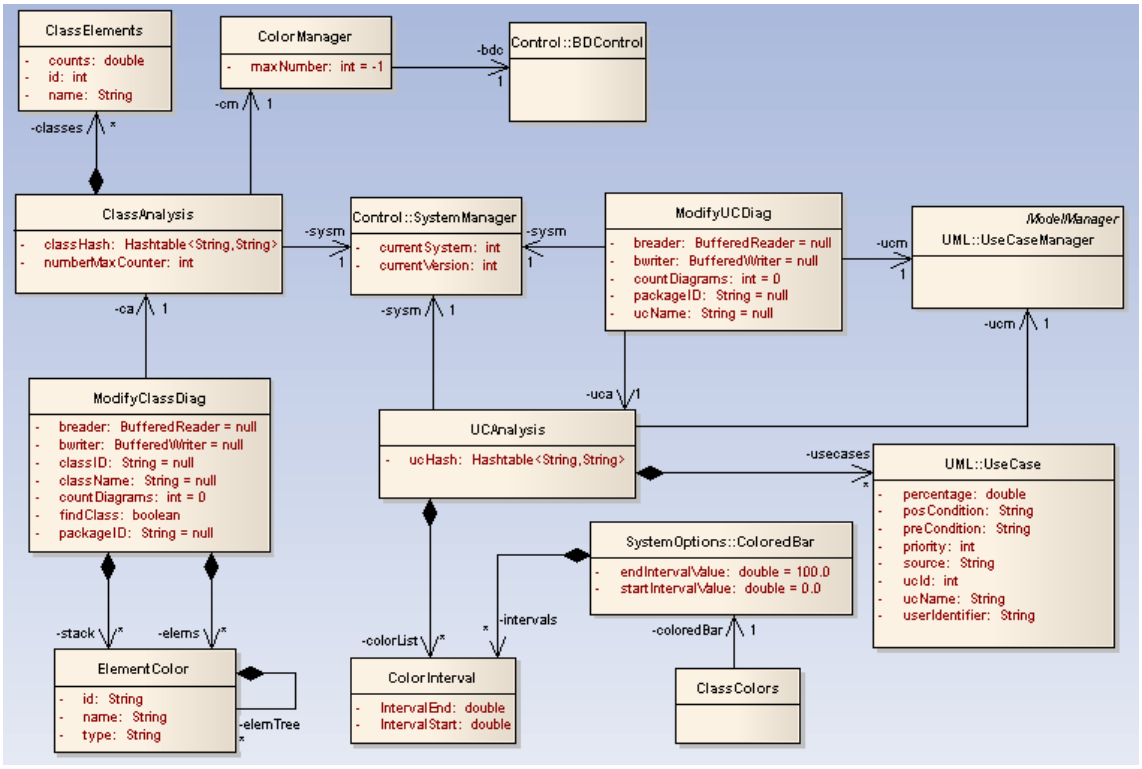


Figura 74. Diagrama de Sequência da execução da exportação do diagrama de casos de uso.

4.3.8.1 *ModifyClassDiag* e *ModifyUCDiag*

Tanto no caso da geração de diagramas de casos de utilização coloridos, como na geração de diagramas de classes coloridos, torna-se necessária a leitura de diagramas externos. Se apenas fossem exportados os resultados para elementos contidos no sistema, o resultado da cobertura seria sempre 100%. Um teste de cobertura que apenas represente o que foi executado, não é certamente um teste com interesse para a maioria dos testadores. No caso do *ReModeler*, os dados obtidos através das capturas dinâmicas são comparados com os dados presentes em diagramas criados de modo estático. Os resultados dessa comparação são depois exportados para um novo diagrama, igual ao importado, mas contendo a informação da coloração dos seus elementos. As classes *ModifyClassDiag* e *ModifyUCDiag* são as classes deste pacote que estão encarregues de fazer a leitura dos diagramas externos de classes e casos de utilização, respectivamente, e de criar os novos já

coloridos. À medida que o ficheiro vai sendo lido, os seus elementos vão sendo interpretados e, dependendo do seu tipo, são invocados os métodos das outras classes, que permitem identificar a sua cobertura. Feito isto, cada elemento volta a ser escrito, agora no novo ficheiro, apenas com a alteração da cor de fundo.

4.3.8.2 *ElementColor*

A classe *ElementColor* representa qualquer elemento que é lido dos ficheiros dos diagramas. Existem elementos que aparecem inseridos dentro de outros, como o caso das classes dentro dos pacotes, que são importantes para a identificação clara do elemento a tratar, como o caso de duas classes que podem ter o mesmo nome, mas encontrarem-se em pacotes diferentes. Por esta razão, esta classe também armazena a hierarquia desses vários elementos.

4.3.8.3 *ColorInterval*

Esta classe representa cada intervalo de percentagem e a respectiva cor. Ela apenas contém três variáveis para o seu efeito:

- `private double IntervalStart` – o início do intervalo,
- `private double IntervalEnd` – o fim do intervalo,
- `private Color color` – a cor correspondente ao intervalo em causa.

4.3.8.4 *ClassColors*

A classe *ClassColor* é usada pela interface gráfica, para guardar no sistema os intervalos de percentagem e a respectiva cor, escolhidos pelo utilizador. Estes dados não são guardados de forma persistente.

4.3.8.5 *UCAnalysis*

A preparação e determinação da cobertura dos elementos para o diagrama de casos de utilização coloridos é realizada pela classe *UCAnalysis*. Inicialmente a classe vai criar os intervalos de cor, da forma mostrada na Figura 75. Depois vai tratar de calcular para cada caso de utilização do sistema, a sua percentagem de capturas, afectando-lhe a cor correspondente à percentagem calculada.

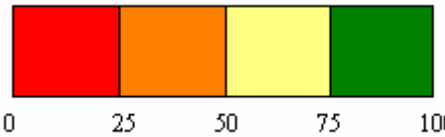


Figura 75. Intervalos de cor e respectivos intervalos de percentagem para os casos de utilização coloridos.

O cálculo da percentagem de cobertura é feito como mostra a Figura 76. Primeiramente são recolhidos todos os casos de utilização existentes no sistema. Para cada um deles são lidos e contabilizados os seus cenários. De seguida é feita uma pesquisa para cada um dos cenários, sobre se já existiu alguma captura no sistema. Caso já tenha ocorrido, é incrementado o valor de uma variável, que no final do ciclo de todos os cenários, é dividida pelo número total dos cenários do caso de utilização.

```
/*
 * scm é instancia de ScenarioManager do pacote UML
 */
public LinkedList<UseCase> prepareUCPercentage(int numVersion, int idSys) {
   LinkedList<UseCase> usecases = getAllUseCases(numVersion, idSys);
   UseCase tempUC = null;
   for(int y = 0; y < usecases.size(); y++){
       tempUC = usecases.get(y);
       LinkedList<Scenario> scenes = getScenarios(tempUC.getUcId());
       int number = scenes.size();
       int capNumber = 0;
       for(int i = 0; i < number; i++)
          if(scm.getAllCaptures(scenes.get(i).getScenarioID()).size() > 0)
                                    capNumber++;
       tempUC.setPercentage(capNumber/number*100);
   }
   return usecases;
}
```

Figura 76. Implementação do método que calcula as percentagens das capturas dos casos de uso.

4.3.8.6 *ClassAnalysis*

De modo análogo à classe *UCAnalysis*, a classe *ClassAnalysis* vai preparar cada elemento do diagrama de classes colorido para ser posteriormente escrito. Mais precisamente, ela vai analisar as classes presentes no sistema e determinar, para cada uma delas, qual a sua percentagem de execução e respectiva cor. Ao contrário do que acontecia na classe anterior, a cor de cada intervalo de percentagem é determinada pelo utilizador e é guardada e disponibilizada no sistema pela classe *ClassColors*. A classe vai usar os métodos disponibilizados pela classe *ColorManager* para realizar a leitura de cada classe do sistema e estabelecer a respectiva percentagem de cobertura, procedendo à comparação dos seus elementos internos executados e dos não executados. No caso do diagrama representar a intensidade de utilização de uma determinada classe, a principal diferença é o cálculo do número de vezes que a

mesma foi executada, que depois é usado para estabelecer a sua percentagem de utilização, dividindo-o pelo número máximo de execuções. A sequência de acções para a determinação da percentagem de intensidade de utilização de uma classe está representada pelo diagrama de sequência UML da Figura 77.

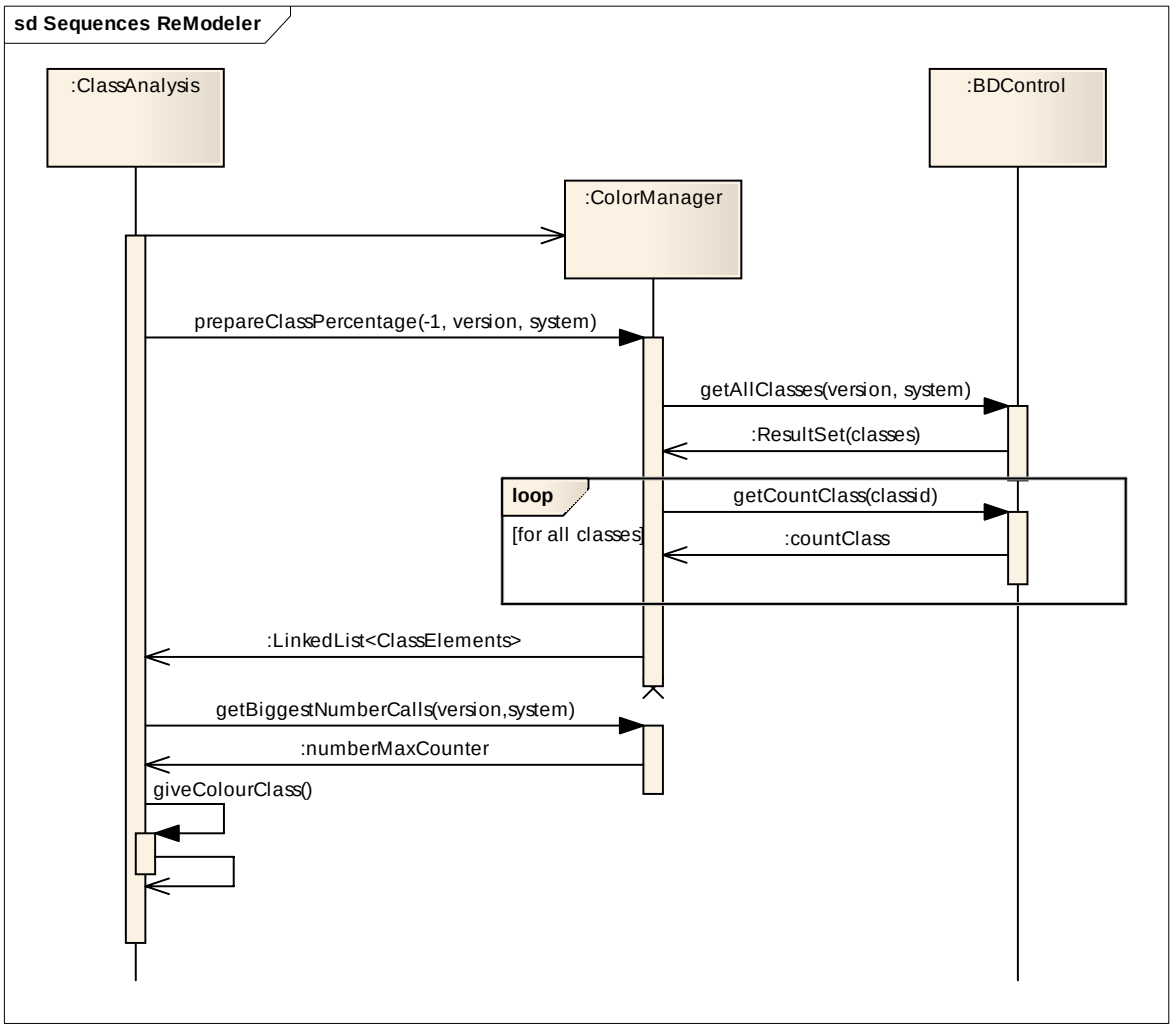


Figura 77. Diagrama de sequência das acções da classe *ClassAnalysis*.

4.3.8.7 *ClassElements*

ClassElements representa cada classe armazenada na base de dados do *ReModeler*, contendo apenas os elementos necessários à sua identificação única no sistema e o valor das invocações que nela foram realizadas.

4.3.8.8 *ColorManager*

Para a produção do diagrama de classes colorido é necessário efectuar algumas leituras dos dados armazenados na base de dados. A classe *ColorManager* disponibiliza um conjunto de métodos, mostrados na Figura 78, que indicam as classes do sistema, o número da contagem do total das invocações de cada uma, o número máximo de invocações a qualquer classe e a percentagem calculada para cada classe. Esta classe vai ser essencialmente usada pela classe *ClassAnalysis* para preparar os elementos para serem escritos no diagrama colorido.

```
//Para uma captura especifica
public int getCountClass(int classId, int capId){…}
//Para todas as capuras já realizadas
public int getCountClass(int classId){…}

public LinkedList<ClassElements> prepareClassPercentage(int capid, int
currentVersion, int currentSystem) {…}

public int getBiggestNumberCalls(int version, int sys){…}
```

Figura 78. Métodos disponibilizados pela classe *ColorManager*.

Capítulo 5

Validação

Conteúdo

Neste capítulo o processo, suportado pela ferramenta desenvolvida, é validado pela aplicação a um caso de estudo

5 Validação

5.1 Introdução Geral

Dada a natureza das técnicas propostas contidas nesta dissertação, a sua validação será efectivada através da aplicação do processo proposto a um sistema real, o qual é descrito na secção 5.2.

5.2 Descrição do Caso de Estudo

Para permitir a replicação do exercício de validação pretendia-se um sistema com o código fonte em Java disponível na web. Por outro lado, para comprovar parcialmente a escabilidade do *ReModeler* pretendia-se um sistema de dimensão média (com umas dezenas de classes). A fonte de procura foi a Internet, mais precisamente o *SourceForge* [SourceForge, 2008] que é um repositório de sistemas de software livre.

Ao fim de algumas pesquisas, o sistema escolhido foi o *SweetHome3D* [eTeks, 2006-2008]. O *SweetHome3D* é uma aplicação de design de interiores, que permite a criação de uma planta de uma casa e a adição de mobiliário à mesma.

Figura 79. Imagem de uma produção da aplicação *SweetHome3D*.

5.2.1 Funcionalidades

A aplicação *SweetHome3D* destina-se a pessoas que desejam planear interiores de um espaço. Em primeira instância, os seus utilizadores podem desenhar paredes,

usando para tal a imagem do plano, de modo a construir salas ou casas completas (Figura 80).

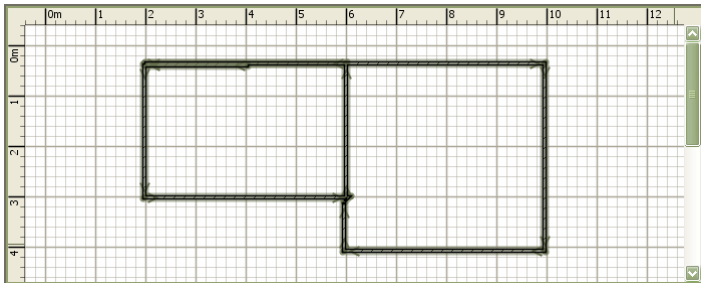


Figura 80. Construção de paredes na vista de plano (2D).

Depois de concluída essa fase, é possível adicionar mobiliário a cada espaço, através de facilidades de *drag and drop*, a partir de um catálogo disponibilizado para o efeito, organizado em categorias.

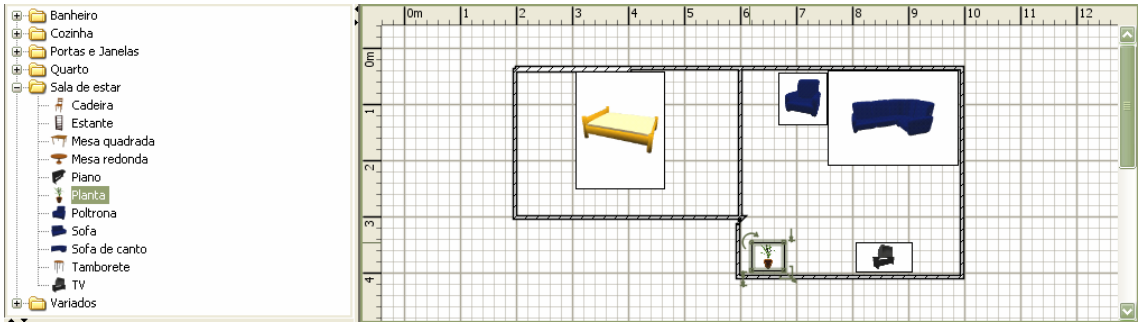


Figura 81. Adição de mobiliário a cada espaço.

Cada peça adicionada pode ser rodada, deslocada ou modificadas as suas dimensões e cores.

Qualquer alteração feita na visão bidimensional (2D) é imediatamente reflectida na vista tridimensional (3D), para que o utilizador consiga ter uma visão mais realista do seu projecto.

A Figura 82 mostra, à esquerda, uma imagem a 3D de um projecto já criado e à direita uma possível utilização do sistema, apresentando um *screenshot* do editor da ferramenta. Neste *screenshot* é possível ver à esquerda o catálogo do mobiliário disponível e um quadro com a listagem do mobiliário já presente no plano, e as respectivas características. À direita, em cima, está a visão 2D do projecto criado, enquanto que em baixo está a representação 3D correspondente.

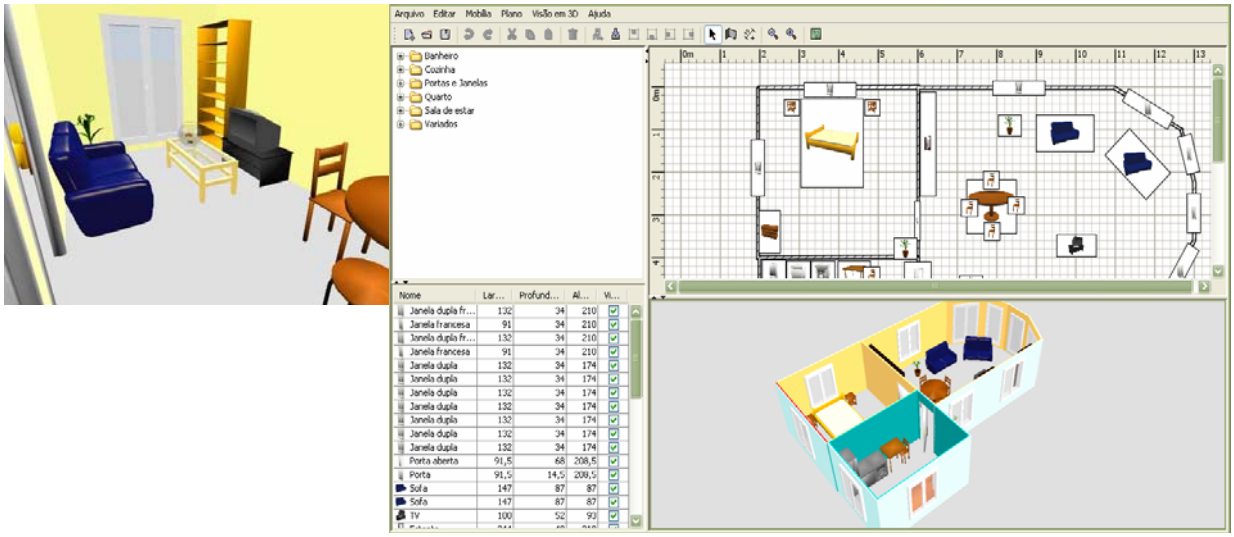


Figura 82. Vista 3D da produção (esquerda) e vista total da aplicação (direita).

Antes do início da validação do *ReModeler*, foram realizadas algumas actividades de engenharia inversa e de documentação do sistema *Sweet Home 3D*, tal como descrito no passo *Model Weaving* do processo do *ReModeler* (capítulo 2). Para o efeito foi utilizada a ferramenta *Enterprise Architect* [SparxSystems, 2008], que permitiu a recuperação do diagrama de classes UML, a partir do código fonte. Com a mesma ferramenta foi ainda criado um diagrama de casos de utilização, que descreve algumas das funcionalidades do sistema *SweetHome3D* (ver Figura 83). Neste último diagrama estão representados dez casos de utilização, que incluem as funcionalidades básicas necessárias à utilização do sistema. Inicialmente é necessário *criar paredes* (*Create Walls*) até construir um espaço a ser decorado. Depois é possível *adicionar mobiliário* (*Add Furniture*), podendo este ser sofrer *rotações* (*Rotate Furniture*), *alterações de posicionamento* (*Move Furniture*) ou mesmo *alterações de tamanho* (*Stretch Furniture*). Para além disto, é possível *salvar o projecto desenvolvido* (*Save Home*), *alterar a vista 3D* (*Change View*) ou *aplicar diferentes zooms* (*Apply Zoom*). Por fim, aparecem as funcionalidades de *acesso à ajuda* (*Provide Help*) e de fornecimento de informação sobre a ferramenta (*Inform About)*.

Ambos os diagramas criados na ferramenta foram exportados em formato XMI para posterior importação e utilização pelo *ReModeler*.

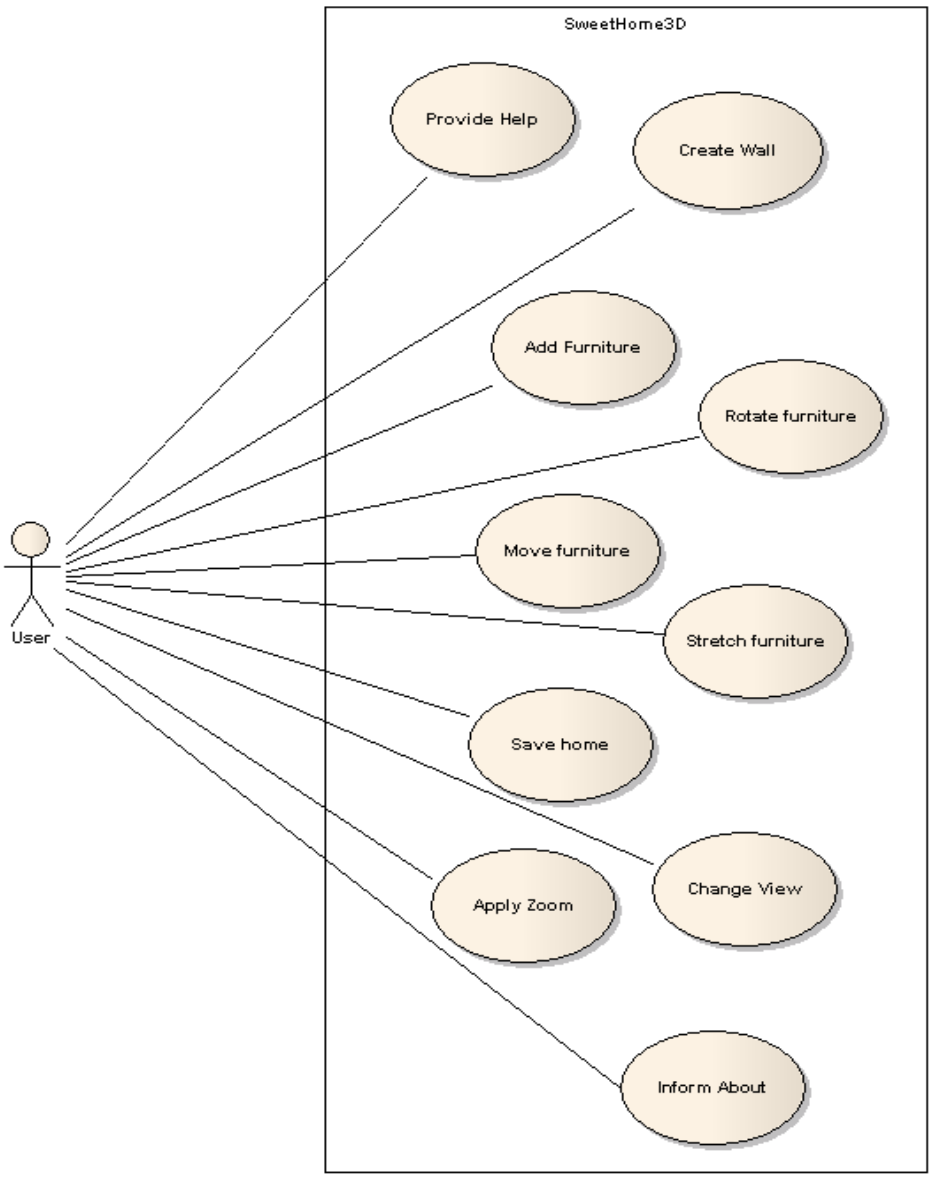


Figura 83. Excerto do diagrama de casos de utilização para o sistema *SweetHome3D*.

5.3 Processo do *ReModeler*

Apesar de existir actualmente um conjunto de ferramentas capazes de produzir parte dos artefactos propostos nesta dissertação (ver anexo B), o modo de produção, os resultados obtidos e a ligação entre eles não se assemelha à apresentada.

O mecanismo de geração dos artefactos propostos está descrito no processo do *ReModeler*, apresentado no capítulo 3 desta dissertação.

5.3.1 Iniciar do *ReModeler*

Após escolhido o sistema a usar, foi necessário adicionar os pacotes que constituem o *ReModeler* ao código fonte do sistema em consideração.

Antes de iniciar a utilização da ferramenta é necessário compilar os sistemas em conjunto, para permitir o entrelaçamento de ambos, que vai posteriormente possibilitar a captura de execução.

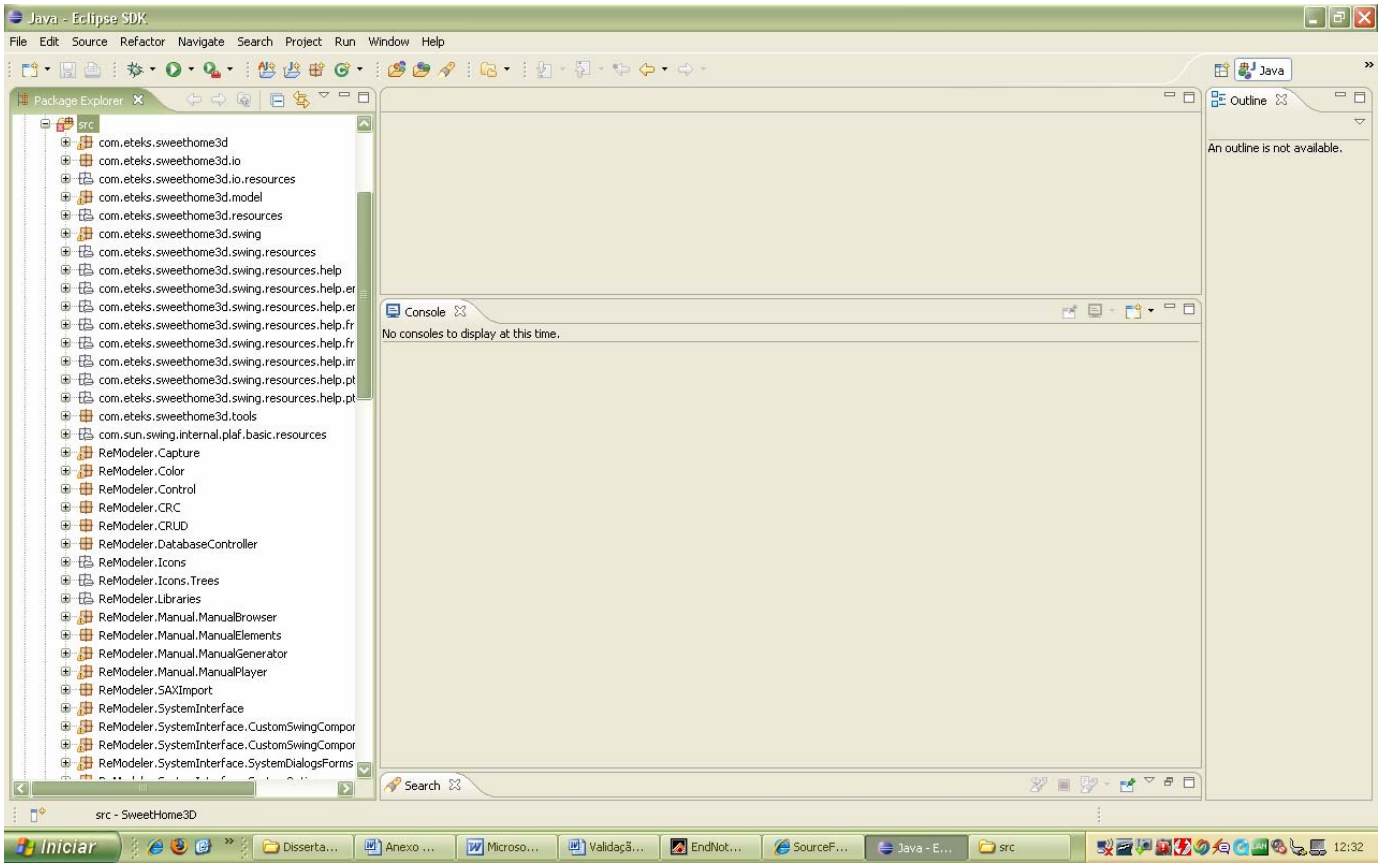


Figura 84. Compilação e execução de ambos os sistemas com conjunto, no Eclipse.

De seguida é executado o sistema em análise, como se de uma utilização normal se tratasse. Os dois passos anteriores foram realizados na plataforma de desenvolvimento Eclipse [Eclipse, 2008], como mostra a Figura 84. No entanto poderiam ter sido realizados em linha de comandos, ou através da execução de um ficheiro de comandos em lote (*Ant file*) da aplicação, como mostra o exemplo da

Figura 85. Neste caso o *ReModeler* está todo num ou mais ficheiros *".jar"* que são adicionados ao projecto alvo.

```
<?xml version="1.0"?>

<project name="ClassEditor" default="compile" >

 <path id="aspect.path">
    <pathelement path="${basedir}/ReModeler.jar"/>
 </path>
    
 <taskdef resource="org/aspectj/tools/ant/taskdefs/aspectjTaskdefs.properties">
    <classpath>
    »    <pathelement location="${basedir}/.../aspectjtools.jar"/>
    </classpath>
 </taskdef>

  <target name="compile" depends="init">
    <iajc sourceroots="."
    »       destDir="${basedir}/bin"
    »    »    classpath="${basedir}/.../aspectjrt.jar"
  »»    debug="on"
    »   <aspectPath refid="aspect.path" />
  </iajc>
  </target>

</project>
```

Figura 85. Exemplo de ficheiro *Ant*.

No ficheiro *Ant* é adicionado o ficheiro "*.jar*" do *ReModeler* aos restantes ficheiros do sistema ("<*path*>…</*path*>") e são adicionadas outras bibliotecas necessárias à execução de aplicações com aspectos. Por fim são introduzidos os comandos de compilação através das *tags* <*target* …><*iajc* …></*iajc*></*target*>.

A primeira janela a ser mostrada é a janela do *ReModeler* que permite escolher a acção que se pretende realizar na sessão iniciada. As opções disponíveis são: *Start program*, em que o sistema em análise é capturado desde a sua inicialização; *Edit Use Cases*, que faz abrir a janela onde o sistema pode ser documentado; *Import Use Cases*, que faz abrir a mesma janela que a opção anterior, mas já dispondo a informação lida de um diagrama de casos de utilização importado; e *Help*, que activa o guia de utilização do *ReModeler*. Nesta validação foi escolhida a opção de iniciar a documentação do sistema (ver Figura 86).

Figura 86. Janela inicial do *ReModeler*.

De seguida é mostrado o editor que vai permitir realizar todas as funcionalidades descritas nesta dissertação, juntamente com as descritas em [Gouveia, 2008].

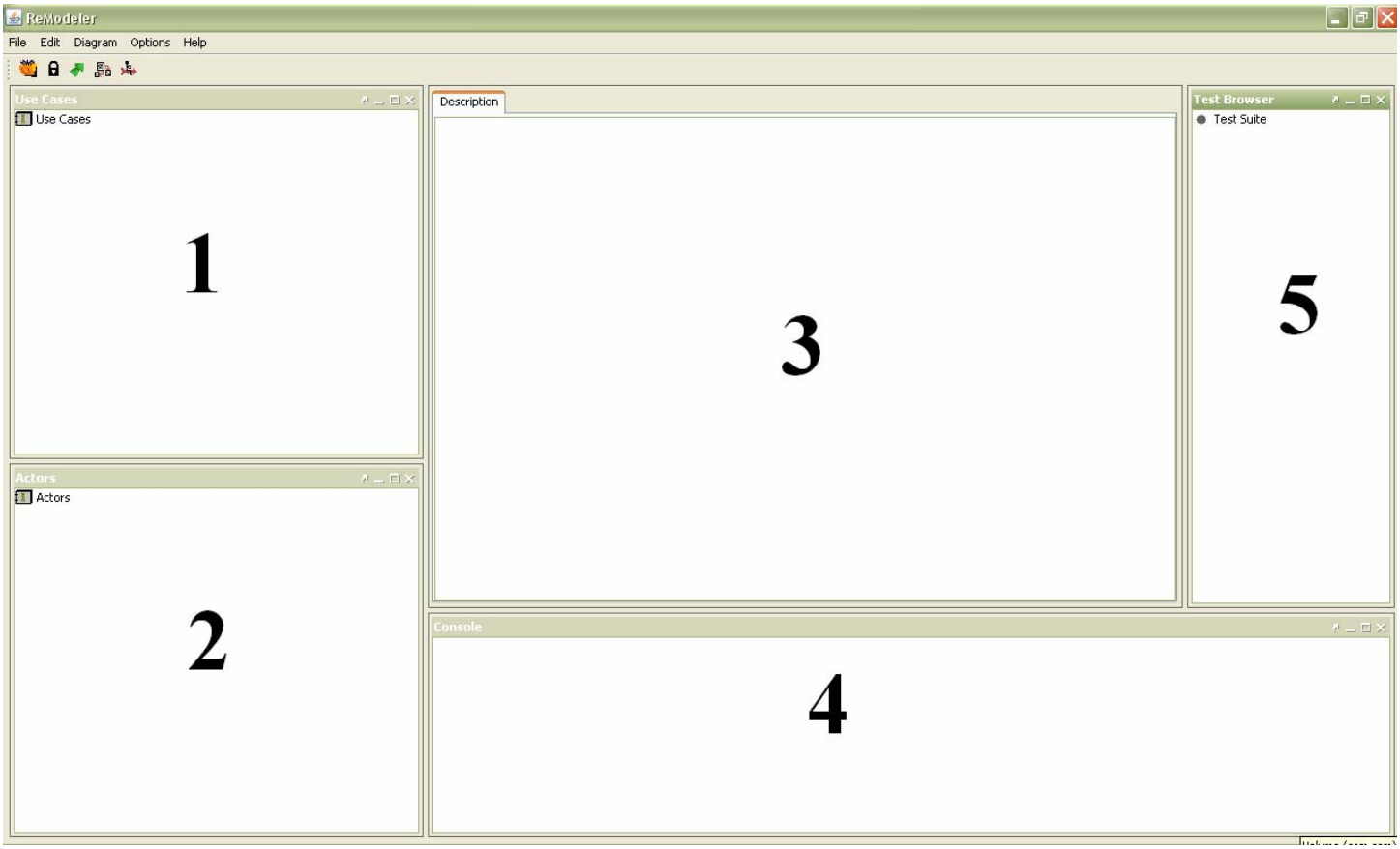


Figura 87. Editor do *ReModeler*.

O editor está organizado por um sistema de cinco janelas flutuantes, como mostra a Figura 87, em que a janela 1 serve para a gestão dos casos de utilização e respectivos cenários, a janela 2 serve para gerir os actores dos casos de utilização, a janela 3 serve para a visualização dos dados e resultados obtidos, a janela 4 serve para apresentar mensagens ao utilizador e, finalmente, a janela 5 serve para gerir os agrupamentos de baterias de testes [Gouveia, 2008].

5.3.2 Documentação do sistema

O primeiro passo a realizar, normalmente efectuado por um analista do negócio ou por um perito do domínio, é a produção da descrição detalhada das funcionalidades do sistema em consideração.

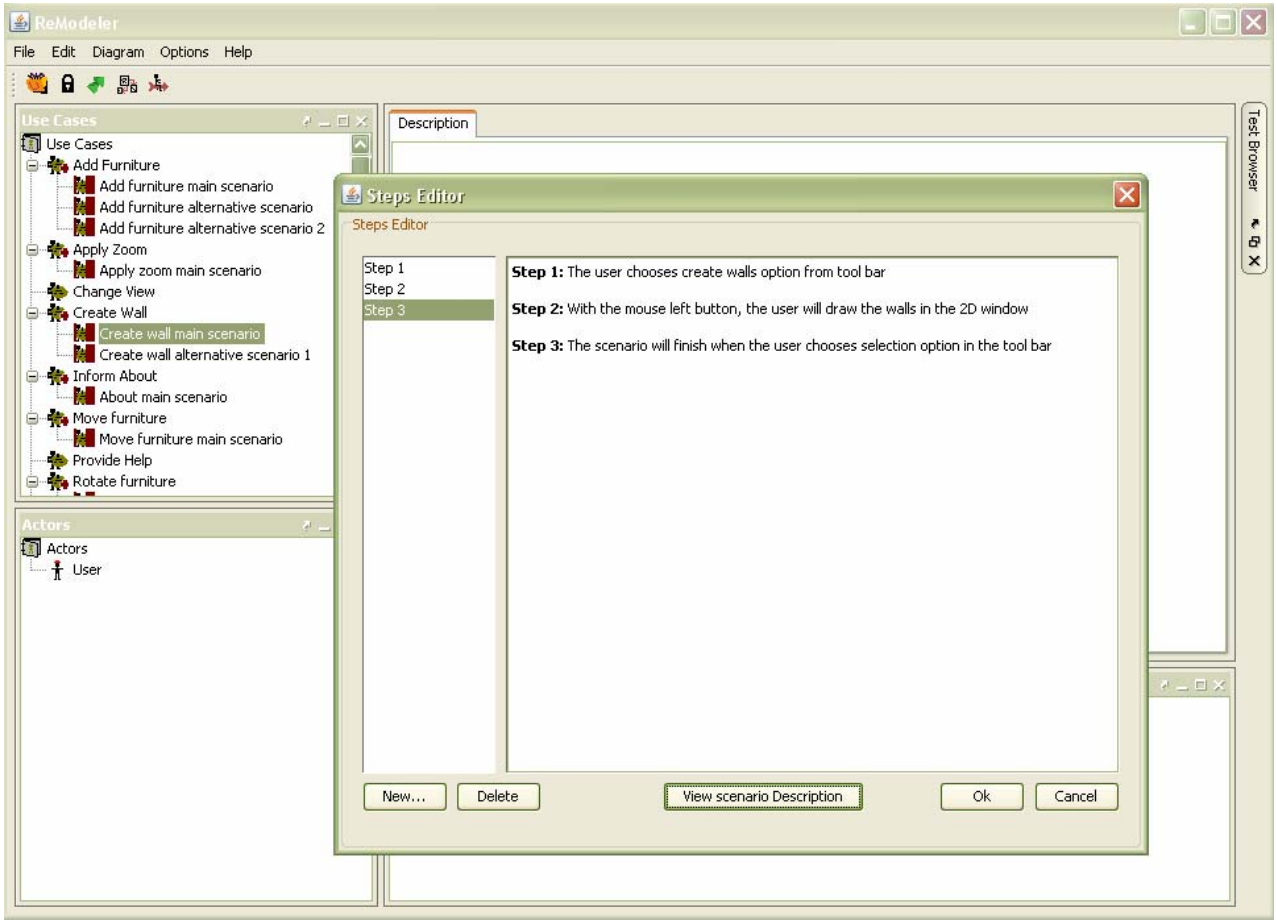


Figura 88. Editor de *steps*.

Para tal, são criados os casos de utilização e respectivos cenários, num editor criado para o efeito. Cada cenário inserido deve ser descrito por um conjunto de passos (*steps*), usando o editor respectivo (ver Figura 88). A documentação do

sistema em análise pode ser facilitada se já existir um diagrama de casos de utilização construído para o mesmo. O *ReModeler* permite a importação de diagramas de casos de utilização em formato XMI, para depois editar a respectiva estrutura de cenários. Neste caso, importou-se o diagrama da Figura 83 de modo a acelerar o processo de documentação do sistema e criaram-se os respectivos cenários e descrições, como mostra a Figura 89.

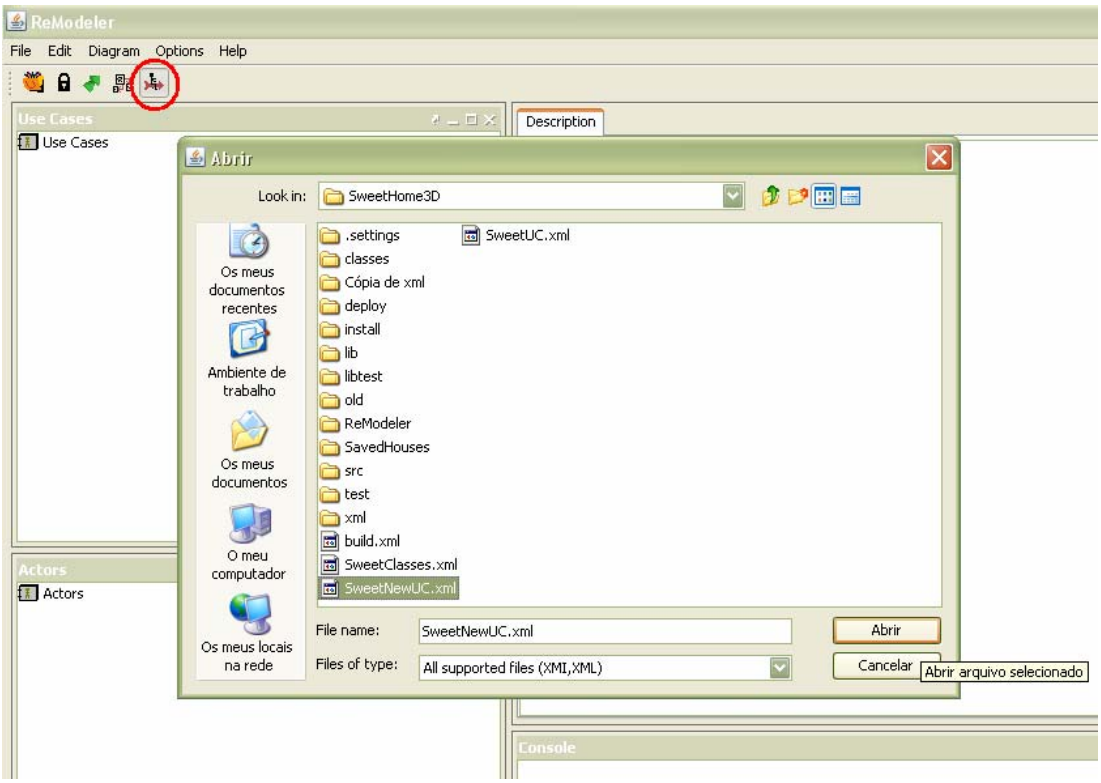


Figura 89. Importação de diagrama de casos de utilização em formato XMI.

A capacidade de definir os cenários dos casos de utilização está ausente na maioria das ferramentas UML. Com efeito estas geralmente suportam apenas a notação diagramática dos casos de utilização, na qual apenas é possível adicionar notas UML.

5.3.3 Captura de um cenário

O passo seguinte a ser efectuado é a execução do *Scenario Capturer*, habitualmente realizado por um perito do domínio.

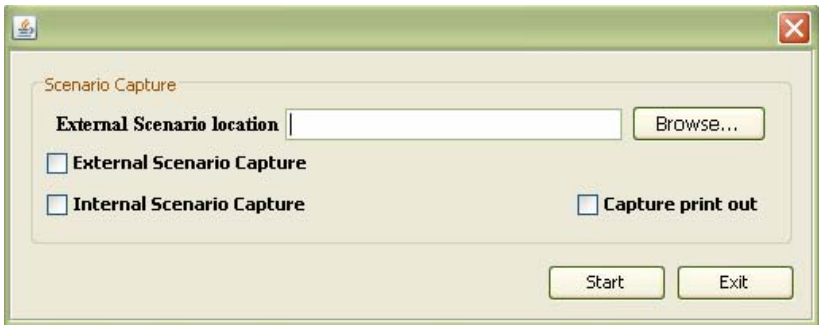


Figura 90. Comando de execução da captura de um cenário.

O *Scenario Capturer* é composto por duas capturas distintas: a captura externa e a captura interna do cenário, que podem no entanto ser executadas simultaneamente (ver Figura 90). A captura externa, que está fora do âmbito desta dissertação, regista todas as interacções do utilizador com o sistema, sob a forma de um filme [Gouveia, 2008]. Para esta validação, apenas são realizadas as capturas internas, para cada cenário separadamente.

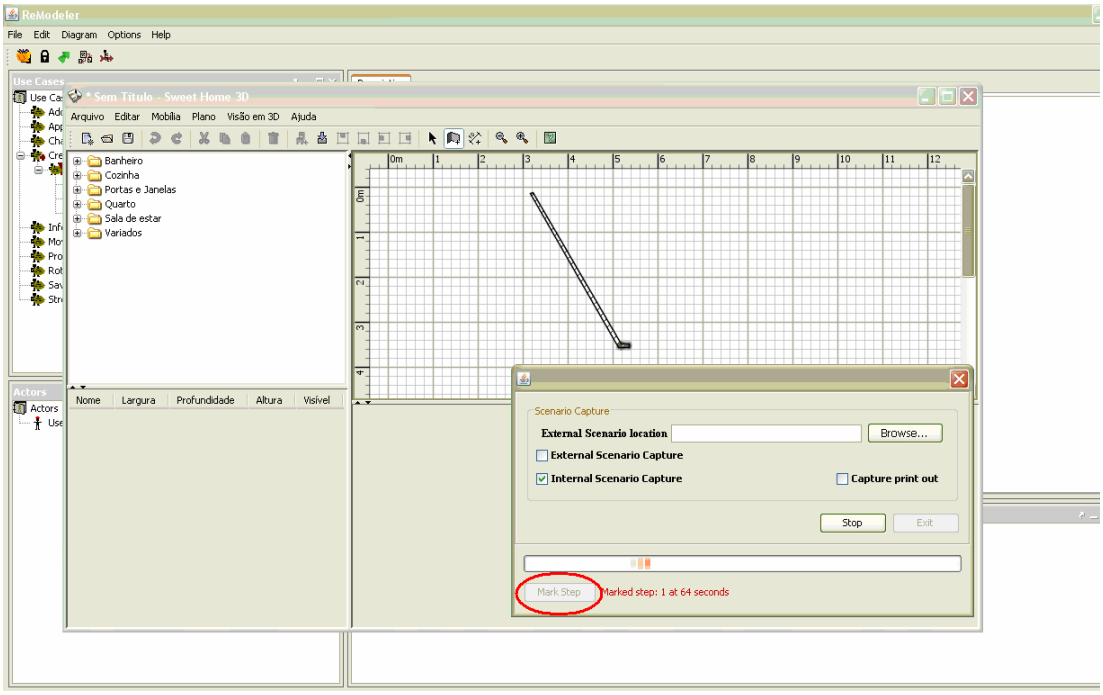


Figura 91. Captura interna do cenário principal de *Create Walls*, em simultâneo com a marcação de um passo.

A captura de cada cenário é realizada passo a passo, como se o sistema estivesse a ser utilizado em prática corrente. Enquanto isto, numa janela separada, deve ser marcado qual dos passos do cenário se está a executar em cada momento (ver Figura 91). Considera-se que os passos são sempre sequenciais (ver secção 5.4) e portanto, para fazer a sua marcação basta carregar no botão "*Mark Step*", visível na Figura 91. Por marcação entenda-se, neste contexto, a explicitação de qual é o passo do cenário que está a ser executado. Actualmente a descrição dos passos dos cenários deverá ter sido realizada previamente, embora idealmente devesse ser possível fazê-la simultaneamente com a captura.

5.3.4 Geração de Diagramas de Sequência e Matriz CRUD simples

O *Scenario Capturer* só produz algum entregável, um filme, quando é feita uma captura externa. Caso contrário, os dados recolhidos na captura são armazenados no repositório persistente do *ReModeler*, para permitirem a geração dos artefactos descritos anteriormente (ver capítulo 2). O primeiro artefacto a gerar é o diagrama de sequência UML, que é automaticamente exportado em formato XMI, para ser importado e visualizado na ferramenta escolhida, o *Enterprise Architect* (ver Figura 92 e Figura 93).

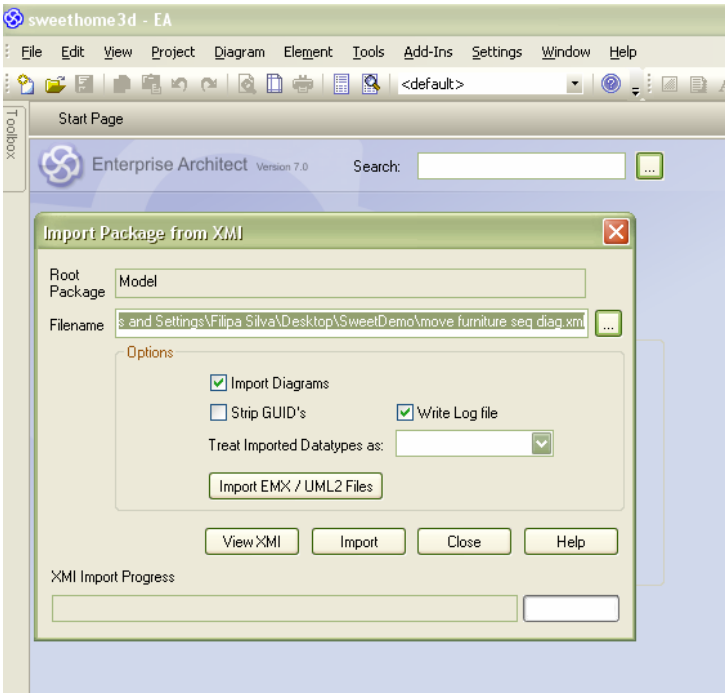


Figura 92. Importação do diagrama de sequência na ferramenta *Enterprise Architect*.

Dado o sistema *SweetHome 3D* ser um sistema de média dimensão (no que respeita ao número de classes que constituem o sistema), a validação dos diagramas gerados torna-se impraticável se não existir um mecanismo que permita seleccionar apenas o que se pretende analisar.

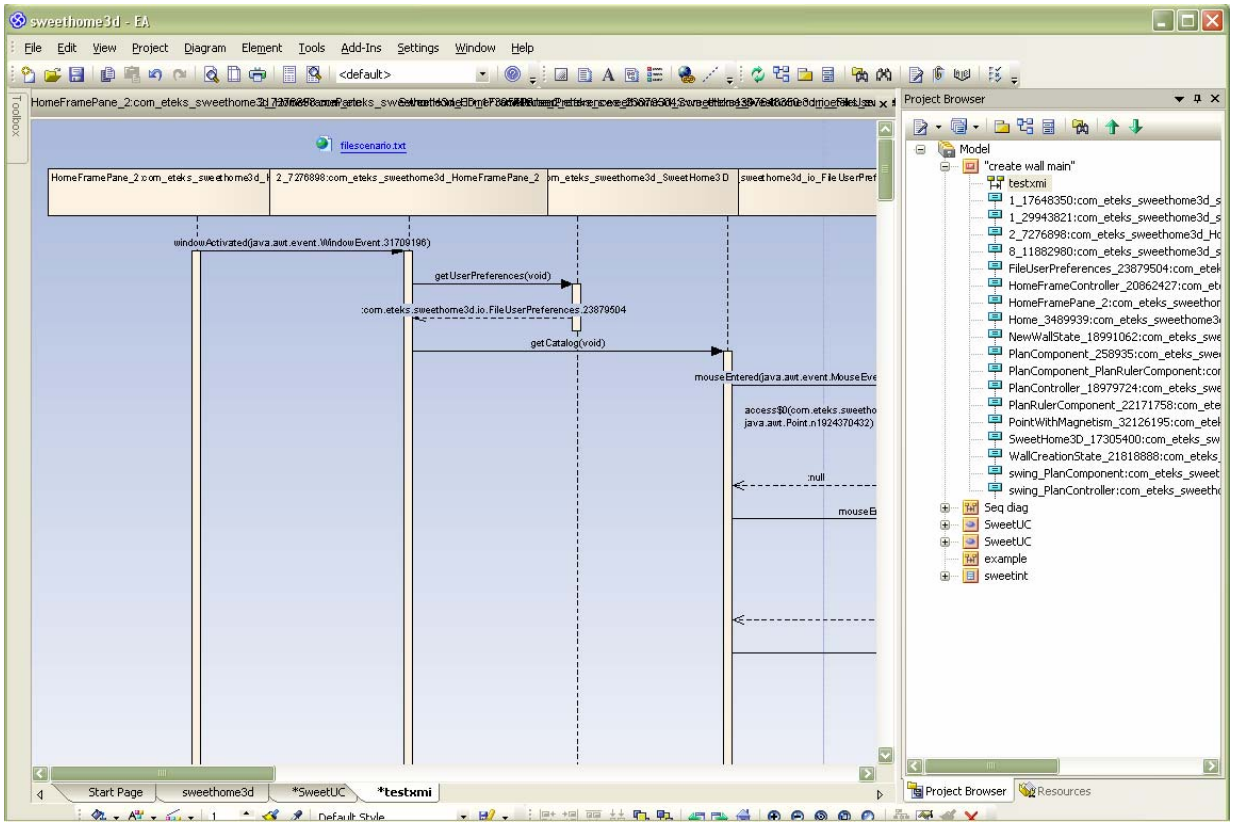


Figura 93. Visualização do diagrama de sequência gerado.

Com efeito, um ficheiro de um diagrama de sequência gerado pode ser bastante grande, mesmo para capturas pequenas. Por exemplo, na captura do cenário *Move furniture main scenario*, em que uma peça de mobília é movida, o ficheiro gerado ficou com cerca de 10220 KB e com 171273 linhas de código, o que implica alguns milhares de mensagens envolvidas. A análise de um ficheiro destes pode ser complicada e até desnecessária, caso o foco da mesma não seja o sistema todo. Por esta razão, o diagrama de sequência gerado vai depender das opções que forem tomadas na janela de filtragem. Quando é solicitada uma geração do diagrama de sequência, é apresentada a janela da Figura 94, que permite escolher (filtrar) os elementos relevantes à análise. Estes elementos correspondem aos métodos, agrupados por classes e pacotes, que estiveram envolvidos na execução do cenário. Cada filtragem pode ainda ser armazenada de modo persistente, para posterior reutilização.

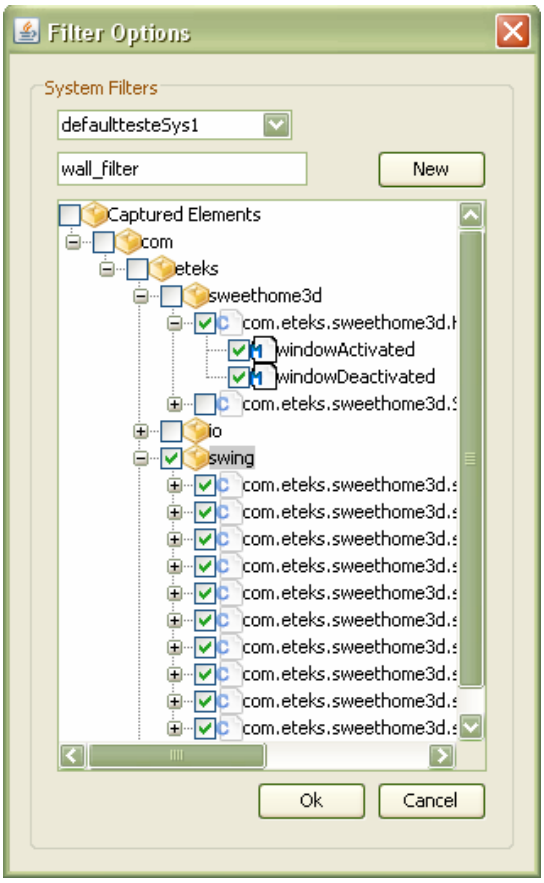


Figura 94. Janela de filtragem de elementos.

Um diagrama de sequência filtrado pode ser visualizado na ferramenta escolhida, sendo que todas as classes filtradas vão ser agrupadas num objecto *Filter* (ver Figura 95).

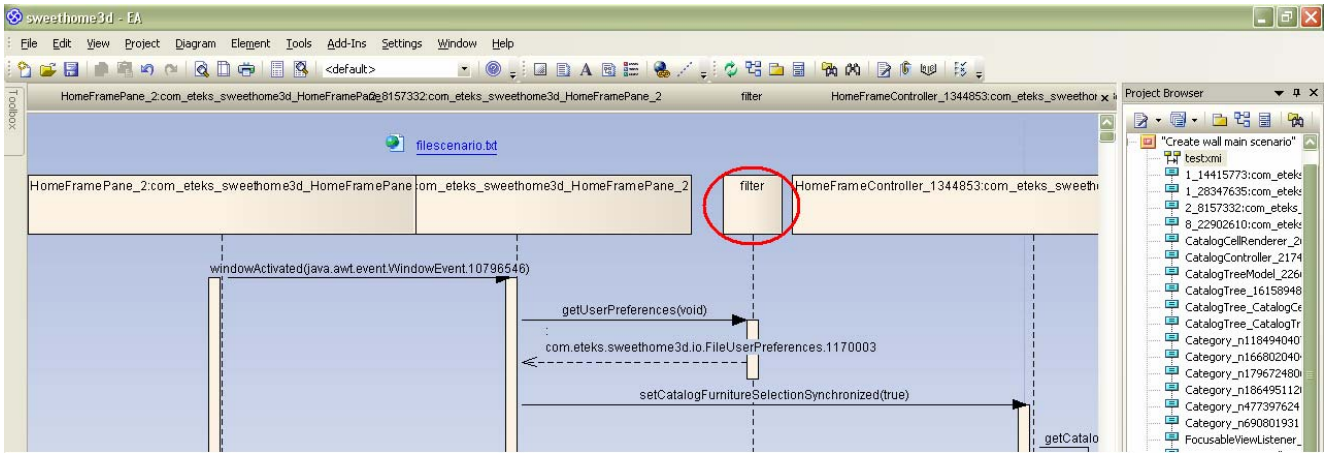


Figura 95. Diagrama de Sequência filtrado.

Quando apenas um método é filtrado, ele deverá ficar omitido do diagrama, como mostra o exemplo das Figura 96, Figura 97 e Figura 98.

```
public void enablePasteAction() {
    HomePane view = ((HomePane)getView());
    if (this.focusedView == getFurnitureController().getView()
        || this.focusedView == getPlanController().getView()
        || this.focusedView == getHomeController3D().getView()){
      boolean selectionMode =
            getPlanController().getMode() ==
               PlanController.Mode.SELECTION;
      view.setEnabled(HomePane.ActionType.PASTE,
          selectionMode && !view.isClipboardEmpty());
    } else {
      view.setEnabled(HomePane.ActionType.PASTE, false);
    }
  }
```

Figura 96. Código fonte do método enablePasteAction().

Na primeira figura é apresentado o código do método *enablePasteAction()* retirado da classe *HomeController*, da qual a classe *HomeFrameController* é uma generalização.

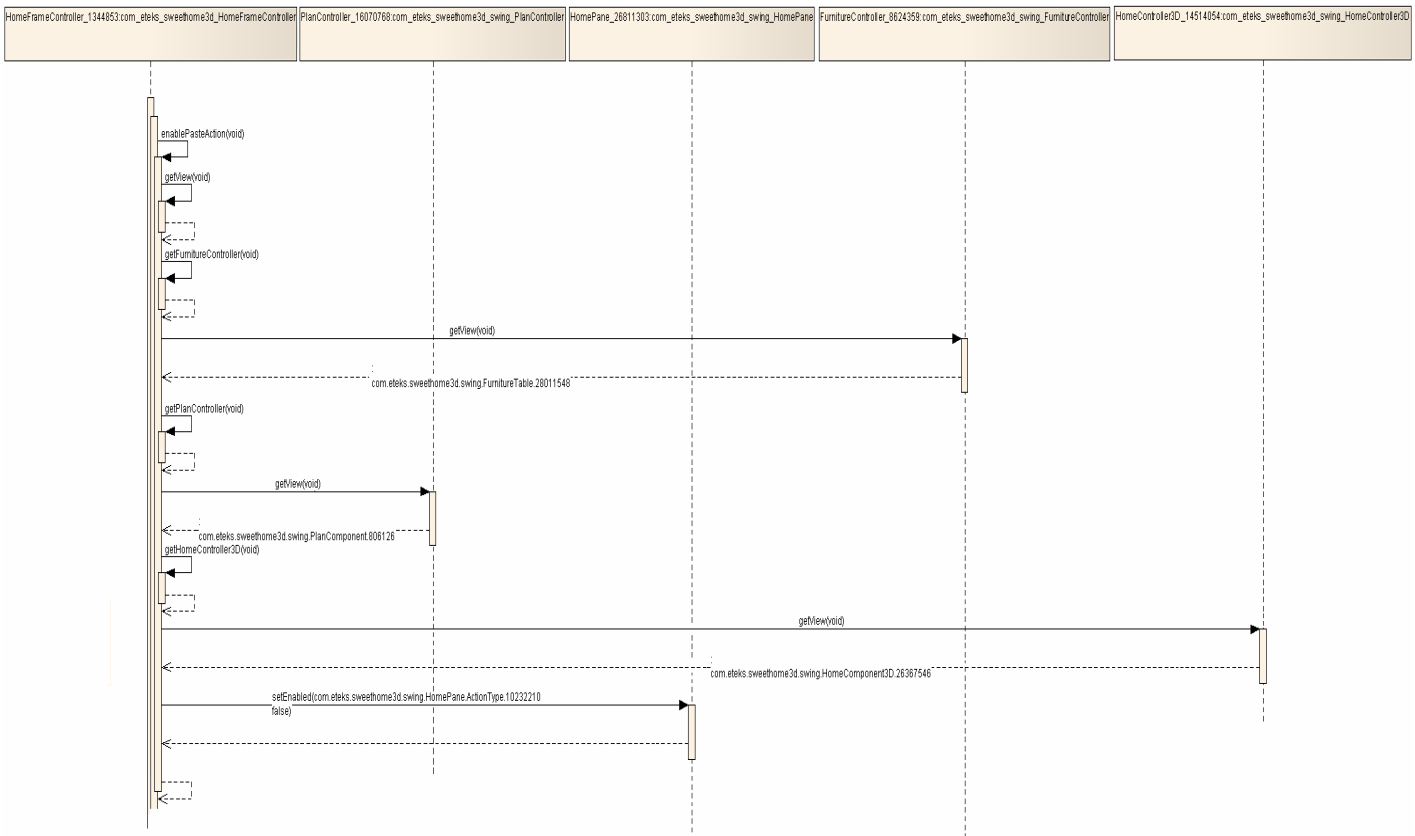


Figura 97. Diagrama de Sequência correspondente à execução do método de *enablePasteAction(void)*.

A Figura 97 mostra um excerto do diagrama de sequência respeitante à captura do método em análise, sem terem sido tomadas nenhumas opções de filtragem. Por sua vez, o diagrama da Figura 98 mostra a mesma captura do método com a filtragem dos métodos *getView()* e *setEnabled(…)*.

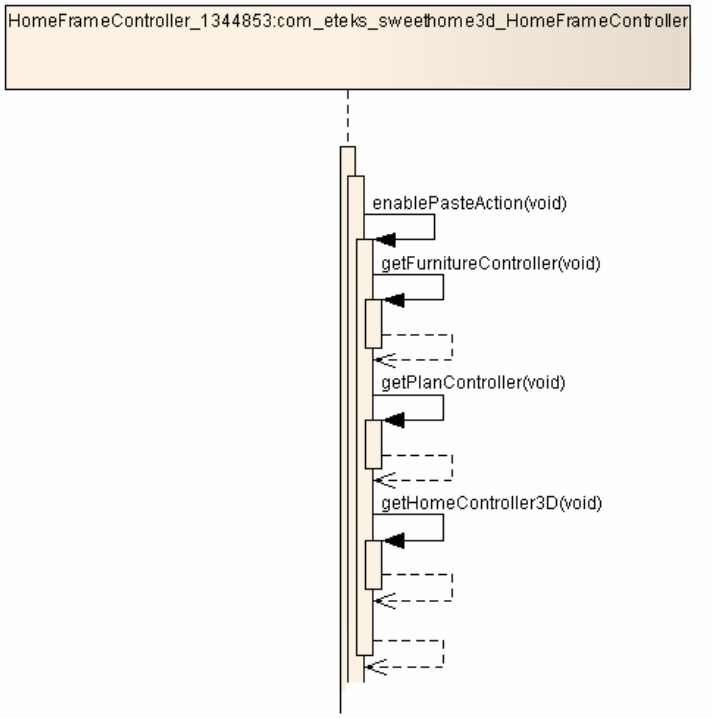


Figura 98. Diagrama de Sequência filtrado correspondente à execução do método de *enablePasteAction(void)*.

Ainda relacionado com o cenário capturado é possível accionar a geração de outro entregável: a matriz CRUD individual. Esta matriz mostra as operações que ocorreram nos elementos envolvidos na captura do cenário seleccionado. Por ser uma matriz gerada em formato *HTML* é possível visualizá-la em qualquer *browser* (ver Figura 99).

Matriz CRUD do cenário Create wall main scenario

Classes / Scenarios	Create wall main scenario
com.eteks.sweethome3d.HomeFrameController	RU
com.eteks.sweethome3d.HomeFramePane$2	U
com.eteks.sweethome3d.model.Camera	RU
com.eteks.sweethome3d.model.CameraEvent	CRD
com.eteks.sweethome3d.model.Category	R
com.eteks.sweethome3d.model.Home	RU
com.eteks.sweethome3d.model.SelectionEvent	CRD
com.eteks.sweethome3d.model.Wall	CR
com.eteks.sweethome3d.model.WallEvent	CRD
com.eteks.sweethome3d.model.WallEvent$Type	CRD
com.eteks.sweethome3d.swing.CatalogController	RU
com.eteks.sweethome3d.swing.CatalogTree	U
com.eteks.sweethome3d.swing.CatalogTree$CatalogCellRenderer	RU
com.eteks.sweethome3d.swing.CatalogTree$CatalogTreeModel	R
com.eteks.sweethome3d.swing.FurnitureController$1	U
com.eteks.sweethome3d.swing.FurnitureTable$17	U
com.eteks.sweethome3d.swing.FurnitureTable$FurnitureTableModel	R
com.eteks.sweethome3d.swing.HomeComponent3D$Wall3D	CRUD
com.eteks.sweethome3d.swing.HomeController$10	U
com.eteks.sweethome3d.swing.HomeController$5	U
com.eteks.sweethome3d.swing.HomeController3D$1	U
com.eteks.sweethome3d.swing.HomeController3D$TopCameraState	RU
com.eteks.sweethome3d.swing.HomePane$1MouseAndFocusListener	U
com.eteks.sweethome3d.swing.HomePane$FocusableViewListener	U
com.eteks.sweethome3d.swing.PlanComponent	RU
com.eteks.sweethome3d.swing.PlanComponent$1	U
com.eteks.sweethome3d.swing.PlanComponent$3	U
com.eteks.sweethome3d.swing.PlanComponent$5	U
com.eteks.sweethome3d.swing.PlanComponent$6	U
com.eteks.sweethome3d.swing.PlanComponent$8	U
com.eteks.sweethome3d.swing.PlanComponent$PlanRulerComponent	RU
com.eteks.sweethome3d.swing.PlanController	RU
com.eteks.sweethome3d.swing.PlanController$2	U
com.eteks.sweethome3d.swing.PlanController$NewWallState	RU
com.eteks.sweethome3d.swing.PlanController$PointWithMagnetism	CRD
com.eteks.sweethome3d.swing.PlanController$WallCreationState	RU

Figura 99. Matriz CRUD gerada para o cenário *Create Wall main scenario*, apresentada num *browser*.

5.3.5 Geração de Diagramas de Casos de Uso coloridos

É possível agrupar os cenários dos casos de utilização em baterias de testes [Gouveia, 2008] e posteriormente avaliar a cobertura de uma dada bateria.

O *ReModeler* permite exportar em formato XMI diagramas de casos de utilização coloridos, em que as cores representam a cobertura, neste caso, a percentagem de todos os cenários que foram executados. Para o conseguir, é necessário importar inicialmente um diagrama de casos de utilização do sistema em formato XMI, como o criado na Figura 83, e em seguida é gerado um ficheiro XMI contendo o diagrama colorido. A visualização deste diagrama é possível importando-o na ferramenta UML escolhida e tirando partido das facilidades de *layout* da mesma (ver Figura 100).

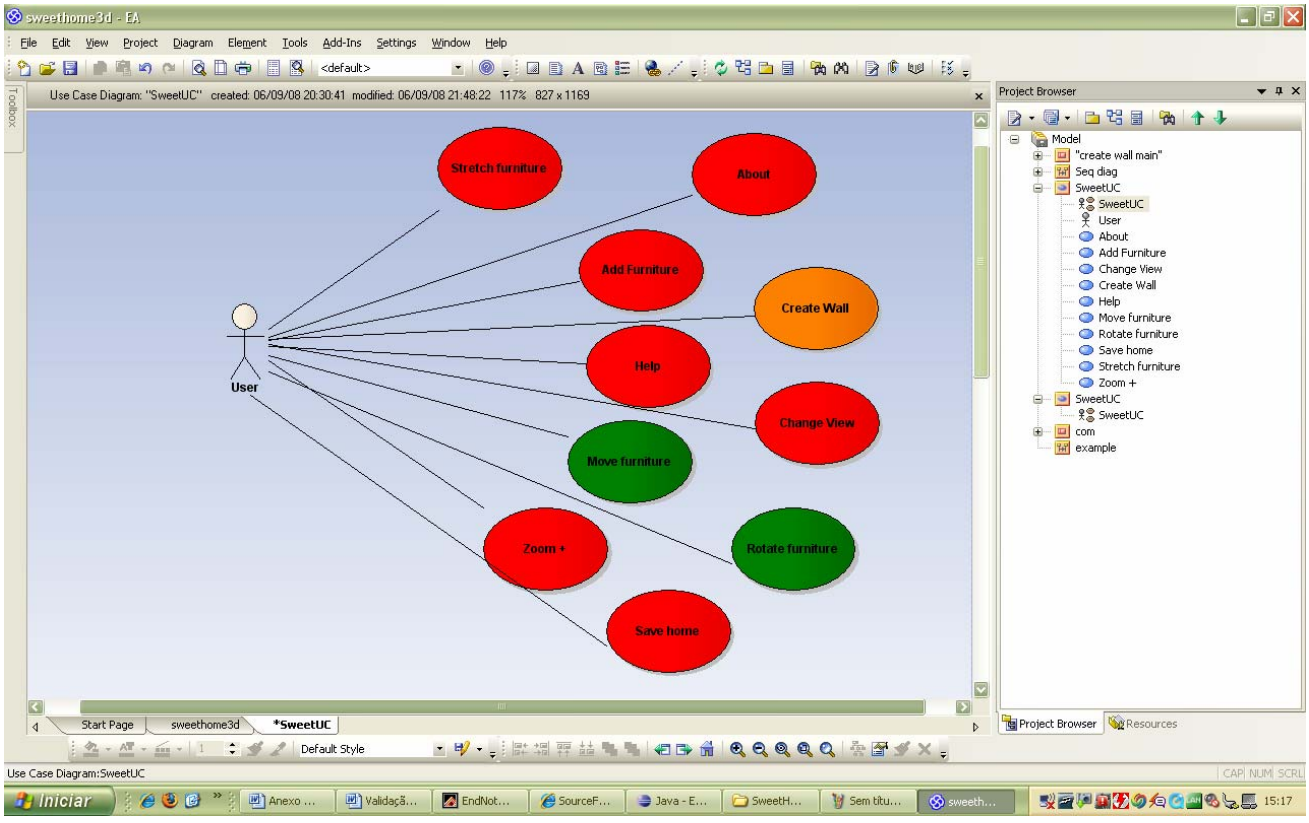


Figura 100. Diagrama de casos de utilização com indicação de cobertura de capturas de cenários.

5.3.6 Geração de Diagramas de Classes coloridos

Para suportar a actividade de testes, é ainda possível gerar artefactos que representem a intensidade e cobertura de execução por cada classe do sistema (ver Figura 101).

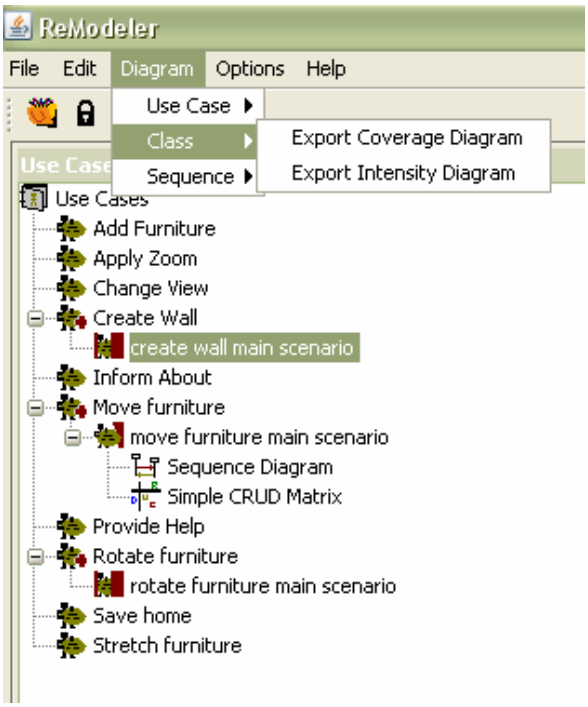


Figura 101. Comandos de exportação de diagramas de classes com indicação de cobertura ou intensidade.

Em ambos os casos, a geração destes artefactos inicia-se com a definição de um gradiente de cores para representar os diferentes intervalos de percentagem (ver Figura 102).

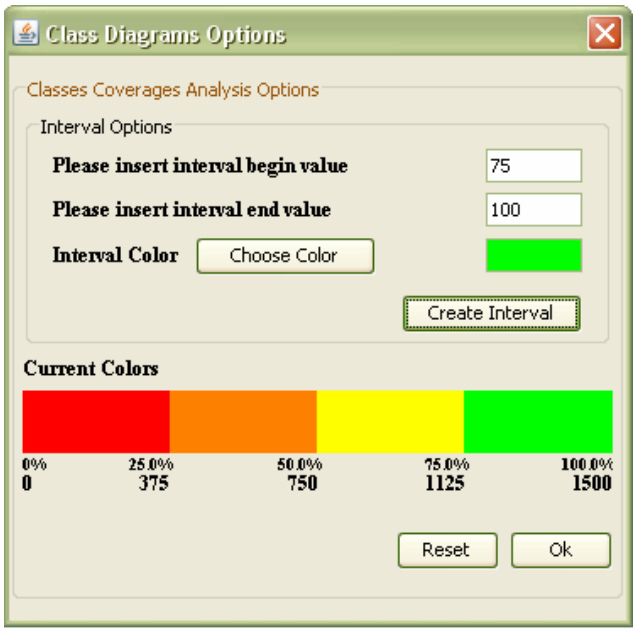


Figura 102. Definição de gradiente de cores para intervalos de percentagem.

Em seguida importa-se o ficheiro XMI que contém a informação da estrutura do sistema, ou seja, o diagrama de classes completo para o sistema em análise. Tal como foi escrito anteriormente, durante o processo de *Model Weaving* foi recuperado e exportado o diagrama de classes do sistema *Sweet Home 3D* com recurso à ferramenta escolhida (ver Figura 103).

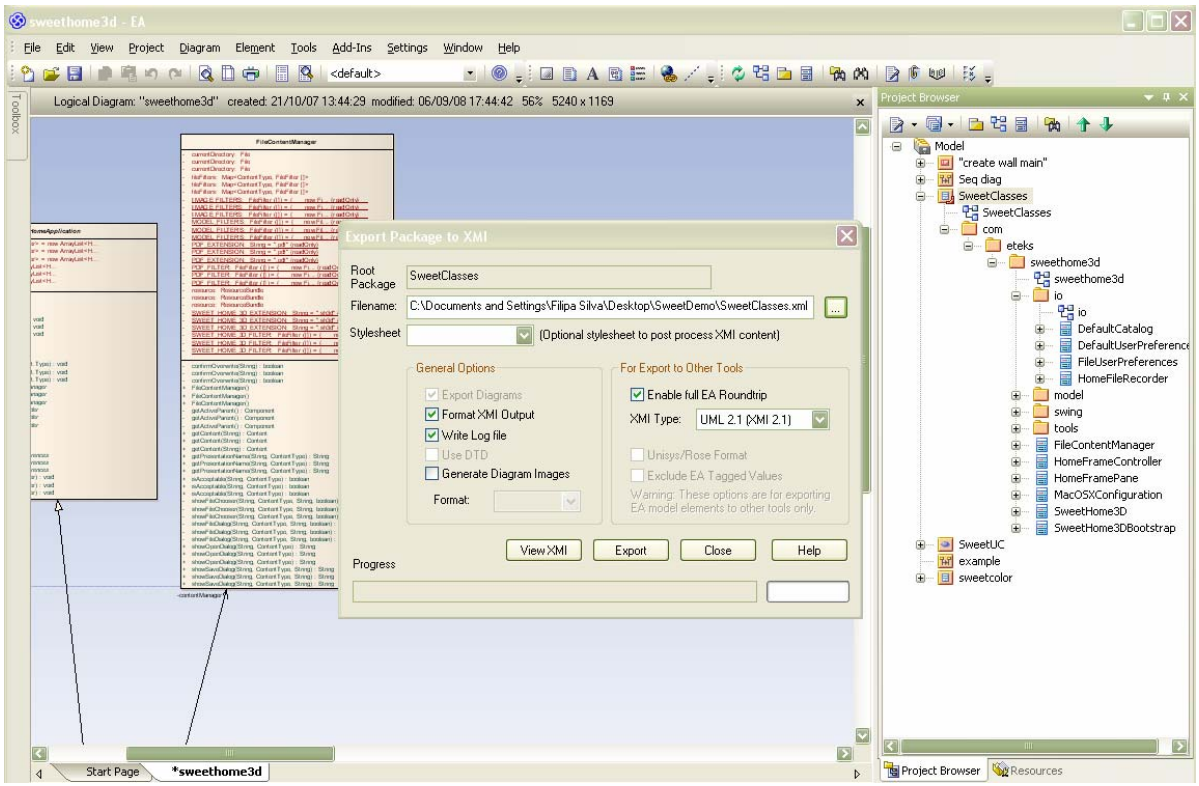


Figura 103. Exportação do diagrama de classes recuperadoem formato XMI, na ferramenta escolhida.

Os diagramas de classes coloridos que resultam destas funcionalidades são também exportados em formato XMI, para depois serem importados na ferramenta *Enterprise Architect* já referida anteriormente. As Figura 104 e Figura 105 mostram excertos dos diagramas de classes, com indicação de cobertura e de intensidade, face às capturas realizadas no sistema, até ao momento da sua geração.

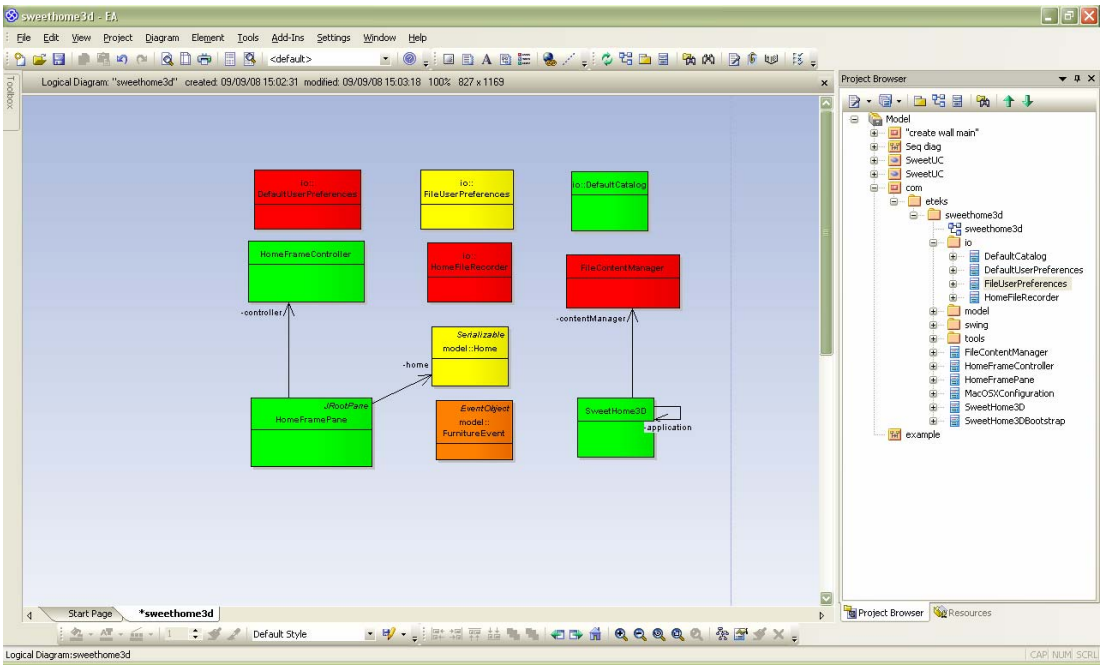


Figura 104. Excerto do diagrama de classes com indicação de cobertura.

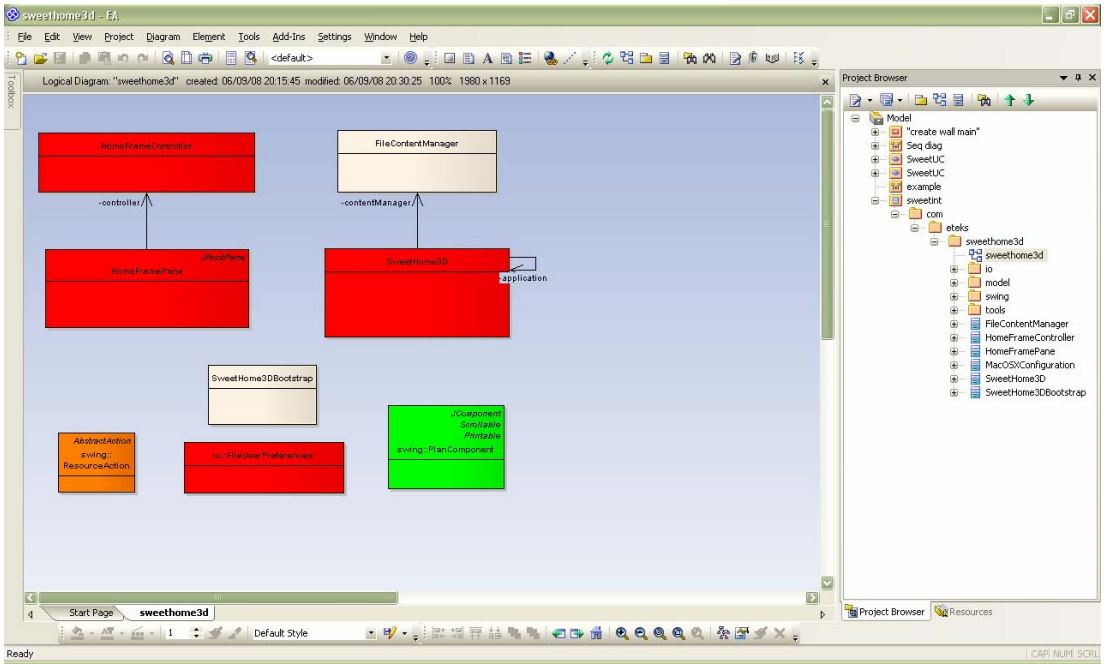


Figura 105. Excerto do diagrama de classes com indicação de intensidade de utilização.

5.3.7 Geração de Matrizes CRUD

De seguida é possível gerar as matrizes CRUD do sistema. Para o conseguir deve ser escolhida a matriz a gerar, através da sua selecção no menu, como mostra a Figura 106, em que é possível escolher que tipo de matriz se pretende.

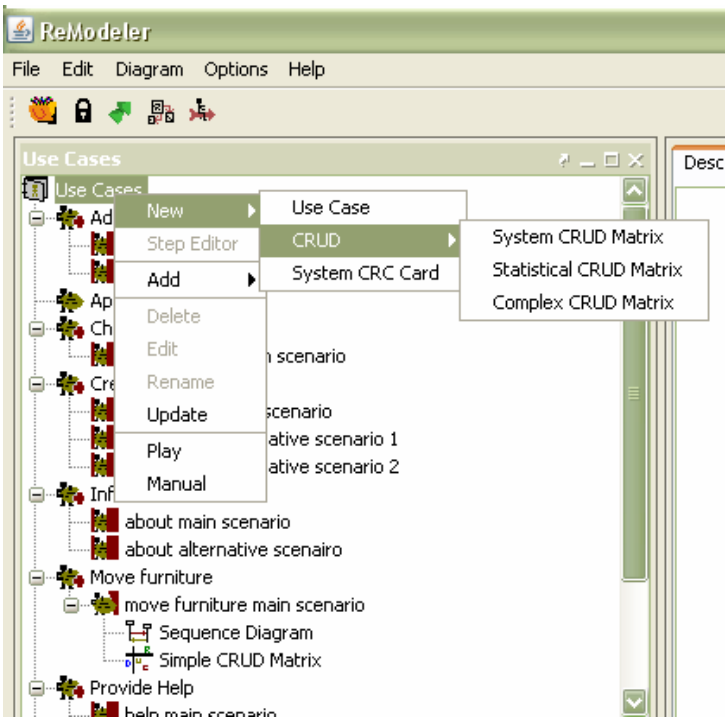


Figura 106. Menu de escolha das matrizes CRUD a gerar.

Para gerar qualquer uma das matrizes CRUD é necessário importar o diagrama de classes do sistema, em formato XMI no *ReModeler*. Caso ainda não tenha sido importado o referido diagrama é mostrada uma janela de aviso, tal a que é mostrada mais adiante na Figura 111.

A primeira matriz a gerar é a matriz de sistema, que mostra todas as classes do sistema e todos os cenários, agrupados por casos de utilização. A Figura 107 mostra um excerto dessa matriz que foi visualizada num *browser*. A matriz representa, de modo sintetizado, que classes são usadas e como, em cada cenário. É de referir que para o caso de utilização *Add Furniture* foram definidos dois cenários, um principal e um alternativo, mas nenhum deles foi ainda capturado. Por esta razão as respectivas

colunas na matriz estão a branco. O mesmo se passa para o caso de utilização *Provide Help*, com a agravante de ainda não terem sido definidos quaisquer cenários.

Matriz CRUD do sistema default

Use Cases	"Add Furniture"		"Create Wall"	"Inform About"	"Move furniture"	"Provide Help"	"Rotate furniture"
Classes/Scenarios	"Add furniture main scenario"	"Add furniture alternative scenario"	"Create wall main scenario"	"About main scenario"	"Move furniture main scenario"		"rotate furniture main scenario"
com.eteks.sweethome3d.HomeFrameController			RU	RU	RU		RU
com.eteks.sweethome3d.HomeFramePane				RU	RU		RU
com.eteks.sweethome3d.HomeFramePane$2			U	U	U		U
com.eteks.sweethome3d.HomeFramePane$5							
com.eteks.sweethome3d.SweetHome3D				R	R		R
com.eteks.sweethome3d.SweetHome3D$10							
com.eteks.sweethome3d.io.DefaultCatalog				RU	RU		RU
com.eteks.sweethome3d.io.FileUserPreferences				R	R		R
com.eteks.sweethome3d.model.Camera			RU		R		
com.eteks.sweethome3d.model.CameraEvent			CRD				
com.eteks.sweethome3d.model.CatalogPieceOfFurniture				R	R		R
com.eteks.sweethome3d.model.Category			R	RU	RU		RU
com.eteks.sweethome3d.model.FurnitureEvent					CRD		
com.eteks.sweethome3d.model.Home			RU	R	RU		RU
com.eteks.sweethome3d.model.HomePieceOfFurniture				R	RU		R
com.eteks.sweethome3d.model.ObserverCamera					R		R
com.eteks.sweethome3d.model.SelectionEvent			CRD				CRD
com.eteks.sweethome3d.model.Wall			CR		R		R
com.eteks.sweethome3d.model.WallEvent			CRD				
com.eteks.sweethome3d.model.WallEvent$Type			CRD				
com.eteks.sweethome3d.swing.CatalogController			RU	RU	RU		RU
com.eteks.sweethome3d.swing.CatalogTree			U	U	U		U
com.eteks.sweethome3d.swing.CatalogTree$CatalogCellRenderer			RU	RU	RU		RU
com.eteks.sweethome3d.swing.CatalogTree$CatalogTreeModel			R	R	R		R
com.eteks.sweethome3d.swing.FurnitureController				R	R		RU
com.eteks.sweethome3d.swing.FurnitureController$1			U				U
com.eteks.sweethome3d.swing.FurnitureTable					U		U

Figura 107. Excerto da Matriz CRUD completa.

A segunda matriz a gerar, a matriz complexa, origina a apresentação de uma janela, idêntica à janela de filtragem, em que se escolhem os elementos e a granularidade dos mesmos (pacote, classe ou método) para introduzir na matriz. Para as escolhas apresentadas na Figura 108, a matriz gerada está representada na Figura 109.

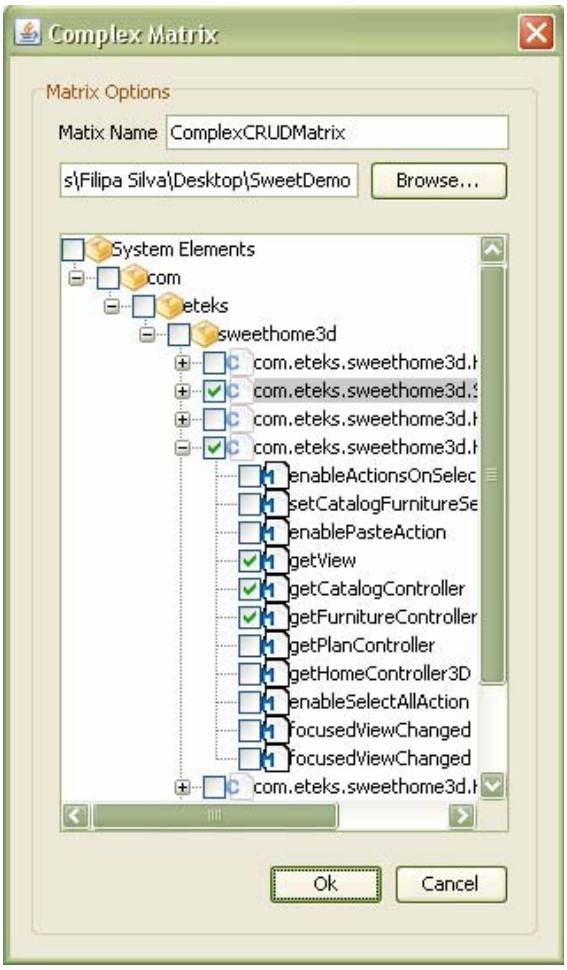


Figura 108. Janela de selecção de elementos a representar na matriz.

Matriz CRUD do sistema default

Use Cases	"Add Furniture"		"Create Wall"	"Inform About"	"Move furniture"	"Provide Help"	"Rotate furniture"
Classes/Scenarios	"Add furniture main scenario"	"Add furniture alternative scenario"	"Create wall main scenario"	"About main scenario"	"Move furniture main scenario"		"rotate furniture main scenario"
com.eteks.sweethome3d.com.eteks.sweethome3d.SweetHome3D				R	R		R
com.eteks.sweethome3d.com.eteks.sweethome3d.HomeFrameController			RU	RU	RU		RU
com.eteks.sweethome3d.com.eteks.sweethome3d.HomeFrameController.getView			R	R	R		R
com.eteks.sweethome3d.com.eteks.sweethome3d.HomeFrameController.getCatalogController			R	R	R		R
com.eteks.sweethome3d.com.eteks.sweethome3d.HomeFrameController.getFurnitureController			R	R	R		R

Generated by ReModeler

Figura 109. Matriz CRUD com granularidade variável.

Por último, pode-se gerar a matriz com mais informação, designada de matriz "estatística", que à matriz de sistema adiciona informação sobre o número de invocações de cada operação (vide Figura 110).

Matriz CRUD do sistema default

Use Cases	"Add Furniture"		"Create Wall"	"Inform About"	"Move furniture"	"Provide Help"	"Rotate furniture"	Totals
Classes/Scenarios	"Add furniture main scenario"	"Add furniture alternative scenario"	"Create wall main scenario"	"About main scenario"	"Move furniture main scenario"		"rotate furniture main scenario"	
com.eteks.sweethome3d.HomeFrameController			RU	RU	RU		RU	C(0) R (220) U(64) D(0)
com.eteks.sweethome3d.HomeFramePane				RU	RU		RU	C(0) R(9) U(10) D(0)
com.eteks.sweethome3d.HomeFramePane$2			U	U	U		U	C(0) R(0) U(17) D(0)
com.eteks.sweethome3d.io.DefaultCatalog				RU	RU		RU	C(0) R(65) U(59) D(0)
com.eteks.sweethome3d.io.FileUserPreferences				R	R		R	C(0) R (827) U(0) D(0)
com.eteks.sweethome3d.model.Camera			RU		R			C(0) R(22) U(3) D(0)
com.eteks.sweethome3d.model.CameraEvent			CRD					C(1) R(1) U(0) D(1)
Totals	C(0) R(0) U(0) D(0)	C(0) R(0) U(0) D(0)	C(25) R (2110) U (627) D (24)	C(5) R (1480) U (249) D (5)	C(8) R (1844) U (350) D(8)	C(0) R (0) U(0) D(0)	C(10) R (3545) U (625) D (10)	

Generated by ReModeler

Figura 110. Excerto da Matriz CRUD com informação sobre a invocação das operações.

5.3.8 Geração de Cartões CRC

Finalmente, falta gerar os cartões CRC para o sistema em análise. Se o diagrama de classes do sistema não tiver sido ainda importado para o *ReModeler*, a activação desta funcionalidade vai originar uma janela de alerta a exigir a sua importação (ver Figura 111).

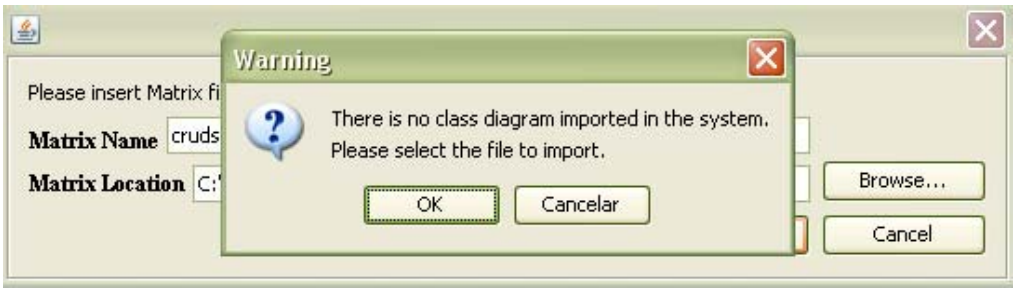


Figura 111. Alerta de importação do diagrama de classes.

Caso contrário, os cartões são gerados como mostram a Figura 112 e a Figura 113. A primeira figura mostra um índice dos cartões gerados que facilita a procura e navegação do cartão a analisar.

CRC Cards for the system default

- The following links can be used to go directly to the CRC cards for the various classes:

- Class com.eteks.sweethome3d.HomeFramePane$2
- Class com.eteks.sweethome3d.SweetHome3D
- Class com.eteks.sweethome3d.io.FileUserPreferences
- Class com.eteks.sweethome3d.HomeFramePane
- Class com.eteks.sweethome3d.io.DefaultCatalog
- Class com.eteks.sweethome3d.HomeFrameController
- Class com.eteks.sweethome3d.swing.CatalogController
- Class com.eteks.sweethome3d.swing.CatalogTree
- Class com.eteks.sweethome3d.swing.PlanComponent$1
- Class com.eteks.sweethome3d.swing.PlanComponent$PlanRulerComponent
- Class com.eteks.sweethome3d.swing.PlanComponent
- Class com.eteks.sweethome3d.swing.PlanController
- Class com.eteks.sweethome3d.swing.PlanController$WallCreationState
- Class com.eteks.sweethome3d.swing.HomePane$FocusableViewListener
- Class com.eteks.sweethome3d.swing.HomePane
- Class com.eteks.sweethome3d.model.Home
- Class com.eteks.sweethome3d.swing.FurnitureController
- Class com.eteks.sweethome3d.swing.HomeController3D
- Class com.eteks.sweethome3d.swing.CatalogTree$CatalogTreeModel
- Class com.eteks.sweethome3d.model.Category
- Class com.eteks.sweethome3d.swing.CatalogTree$CatalogCellRenderer
- Class com.eteks.sweethome3d.swing.PlanController$NewWallState
- Class com.eteks.sweethome3d.model.SelectionEvent
- Class com.eteks.sweethome3d.swing.FurnitureTable
- Class com.eteks.sweethome3d.swing.FurnitureTable$FurnitureTableModel
- Class com.eteks.sweethome3d.swing.FurnitureController$1
- Class com.eteks.sweethome3d.swing.PlanComponent$5
- Class com.eteks.sweethome3d.swing.HomeController
- Class com.eteks.sweethome3d.swing.HomePane$1MouseAndFocusListener
- Class com.eteks.sweethome3d.swing.HomePane$11
- Class com.eteks.sweethome3d.swing.PlanController$PointWithMagnetism
- Class com.eteks.sweethome3d.model.Wall
- Class com.eteks.sweethome3d.model.WallEvent$Type
- Class com.eteks.sweethome3d.model.WallEvent
- Class com.eteks.sweethome3d.swing.HomeController3D$1
- Class com.eteks.sweethome3d.swing.HomeController3D$TopCameraState
- Class com.eteks.sweethome3d.model.Camera
- Class com.eteks.sweethome3d.model.CameraEvent
- Class com.eteks.sweethome3d.swing.PlanController$2
- Class com.eteks.sweethome3d.swing.HomeController$10
- Class com.eteks.sweethome3d.swing.HomeComponent3D
- Class com.eteks.sweethome3d.swing.HomeComponent3D$ObjectBranch
- Class com.eteks.sweethome3d.swing.HomeComponent3D$Wall3D

Figura 112. Excerto do índice de classes que correspondem aos Cartões CRC para o sistema em análise.

Para cada entrada do índice é possível navegar através de *links* para o respectivo cartão (ver Figura 113).

Class com.eteks.sweethome3d.HomeFrameController

Responsibilities	Collaborators
"Create wall main scenario"	com.eteks.sweethome3d.HomeFrameController com.eteks.sweethome3d.io.DefaultCatalog com.eteks.sweethome3d.io.FileUserPreferences com.eteks.sweethome3d.model.Home com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.FurnitureController com.eteks.sweethome3d.swing.HomeController3D com.eteks.sweethome3d.swing.HomePane com.eteks.sweethome3d.swing.PlanController
"About main scenario"	com.eteks.sweethome3d.HomeFrameController com.eteks.sweethome3d.io.DefaultCatalog com.eteks.sweethome3d.io.FileUserPreferences com.eteks.sweethome3d.model.CatalogPieceOfFurniture com.eteks.sweethome3d.model.Home com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.FurnitureController com.eteks.sweethome3d.swing.HomePane com.eteks.sweethome3d.swing.PlanController
"Move furniture main scenario"	com.eteks.sweethome3d.HomeFrameController com.eteks.sweethome3d.io.DefaultCatalog com.eteks.sweethome3d.io.FileUserPreferences com.eteks.sweethome3d.model.CatalogPieceOfFurniture com.eteks.sweethome3d.model.Home com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.FurnitureController com.eteks.sweethome3d.swing.HomePane com.eteks.sweethome3d.swing.PlanController
"Rotate furniture main scenario"	com.eteks.sweethome3d.HomeFrameController com.eteks.sweethome3d.io.DefaultCatalog com.eteks.sweethome3d.io.FileUserPreferences com.eteks.sweethome3d.model.CatalogPieceOfFurniture com.eteks.sweethome3d.model.Home com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.FurnitureController com.eteks.sweethome3d.swing.HomePane com.eteks.sweethome3d.swing.PlanController

Class com.eteks.sweethome3d.swing.CatalogController

Responsibilities	Collaborators
"Create wall main scenario"	com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.CatalogTree
"About main scenario"	com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.CatalogTree
"Move furniture main scenario"	com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.CatalogTree
"Rotate furniture main scenario"	com.eteks.sweethome3d.swing.CatalogController com.eteks.sweethome3d.swing.CatalogTree

Figura 113. Exemplos de Cartões CRC para o sistema em análise.

5.4 Ameaças à validação

Neste capítulo foi realizada uma aplicação prática do *ReModeler* a um caso de estudo, de modo a validar o processo proposto nesta dissertação. No entanto, reconhece-se algumas limitações que constituem ameaças para essa validação.

A primeira ameaça a referir é a falta de comparação factual do esforço necessário para realizar as actividades propostas, sem e com o *ReModeler*. Isto deveu-se à dificuldade em contactar com um ambiente empresarial que permitisse tirar indicadores dos custos necessários à realização das tarefas propostas, usando a metodologia seguida internamente em detrimento com o uso do *ReModeler*. No entanto, acredita-se na melhoria do esforço e recursos reais necessários devido ao facto de as gerações de artefactos serem praticamente todas automatizadas.

No que toca à utilização do *ReModeler* existem alguns factores que podem também representar ameaças à validação. Em primeiro lugar, o facto de serem consideradas sequenciais as descrições dos cenários, correspondem a uma situação ideal, não realista. A utilização de ciclos, pontos de extensão e inclusões são algumas das situações previstas na UML e comuns nas descrições realistas, que não estão previstas nesta implementação do *ReModeler*. Por outro lado, o processo de captura da execução dos cenários dos sistemas em análise também apresenta algumas ameaças. A primeira dessas limitações é imposta pelo próprio AspectJ, que não permite o weaving em alguns tipos de sistemas, como sistemas de grande dimensão ou com dimensões excessivas de classes ou métodos. Isto limita a utilização do *ReModeler* a sistemas que estejam bem construídos e/ou que possam ser desacoplados em componentes separados. No entanto, mesmo quando o weaving é possível, existem situações em que o desempenho do sistema em análise com o *ReModeler* decai grandemente, tornando os resultados da sua aplicação nem sempre completamente fiáveis. Isto acontece essencialmente devido ao grande volume de informação com que o *ReModeler* tem de lidar, nomeadamente na escrita e leitura das mensagens capturadas da base de dados. Não obstante, os sistemas multithreading também contribuem para esta perda de desempenho. Mensagens oriundas de execuções concorrenciais (*threads*) levam por vezes a inconsistências na base de dados que despoletam erros durante a execução e geração de artefactos. Esta representa uma ameaça real à validade do *ReModeler*, ainda mais porque

actualmente uma grande parte dos sistemas existentes usa de alguma forma mecanismos de *multithreading*.

Por último, o grande volume de informação recolhido de cada captura vai reflectir-se nos diagramas de sequência gerados. Estes diagramas chegam a atingir uma tal dimensão que nem a ferramenta escolhida tem capacidade para os editar. O mecanismo de filtragem implementado veio melhorar significativamente este problema, mas inibe a visualização de todo o comportamento. Torna-se necessária a implementação de técnicas de compactação dos diagramas, através do reconhecimento de ciclos, o que vem representar numa ajuda fulcral à resolução desta ameaça.

Capítulo 6

Revisão do estado da arte

Conteúdo

Este capítulo fornece uma panorâmica sobre o trabalho que tem sido desenvolvido nas áreas envolventes a esta dissertação.

6 Revisão do estado da arte

6.1 Introdução geral

Neste capítulo descreve-se o trabalho relacionado com o descrito nesta dissertação, em quatro aspectos distintos: a geração automática de diagramas de sequência, de cartões CRC, de matrizes CRUD e de diagramas coloridos relativos à analise de cobertura.

Para melhor situar e comparar o trabalho relacionado entre si e identificar as lacunas existentes, propõe-se seguidamente duas taxionomias, para os dois primeiros aspectos. Cada taxionomia inclui um conjunto de características, através das quais guio o meu estudo comparativo, que servem também para esquematizar as conclusões sob a forma de uma escala ordinal. As características usadas servirão para identificar os pontos fortes e os fracos de cada artigo.

Cada proposta é então apresentada através de uma linha esquemática contendo os resultados da análise taxionómica, do objectivo, do resumo técnico e de uma crítica.

6.2 Framework para a caracterização de mecanismos de captura de Diagramas de Sequência

No decorrer da pesquisa do trabalho relacionado sobre capturas de diagramas de sequência, foram identificados alguns critérios específicos que orientaram a análise sobre o relacionamento entre os vários trabalhos em curso e o que é proposto nesta dissertação. Esses critérios tiveram em consideração os vários mecanismos de recolha de dados de sistema, representação, exportação e visualização da mesma. Também se achou importante referir o contexto de cada estudo, porque esse vai orientar as várias técnicas e opções de implementação propostas.

Existem critérios que são apresentados segundo escalas ordinais, enquanto que outros são representados por escalas nominais, que são explicados a seguir.

6.2.1 Mecanismo de instrumentação

Os mecanismos de instrumentação estão muito ligados ao tipo de captura que se está a referir. O modo de instrumentação no caso de capturas estáticas está fora do âmbito deste estudo. A captura dinâmica refere-se à captura de fluxos de controlo, ou mais precisamente, à sequência de interacções de um sistema em execução. Para capturar essas interacções é necessário criar um mecanismo de intercepção temporal de mensagens da execução de um programa. Estes mecanismos podem ser divididos em instrumentação da aplicação e do ambiente de execução. Em baixo estão definidos quais os mecanismos que foram tomados em consideração, segundo uma escala ordinal.

1. **Manual:** a instrumentação de código manual é a maneira mais arcaica de implementar um mecanismo de intercepção de mensagens. Este método prevê a modificação manual do código de partes ou de um sistema completo. Sempre que se pretende ajustar a captura é necessário modificar o código introduzido. O método de introdução manual de código, para além de complexo e demorado, principalmente para sistemas de média e grande dimensão, é extremamente intrusivo e potencia a ocorrência de erros.

2. **JVM:** existe a opção de instrumentar o ambiente de execução, em vez de instrumentar a aplicação em si. Nos sistemas Java, por exemplo, a instrumentação do ambiente de execução significa a instrumentação da *Java Virtual Machine*.

3. **Debugger:** esta opção representa a possibilidade de construir um debugger específico ou adaptar um preexistente para o efeito de captura. Esta opção é mais acessível que a anterior, mas continua a apresentar complexidade na sua produção e algumas limitações na informação capturada.

4. ***Aspect weaving*:** a programação orientada a aspectos suporta a modularização de *cross-cutting concerns* com aspectos e especificações de *weaving*. Em particular o *AspectJ* disponibiliza um mecanismo de instrumentação não intrusivo, com baixo impacto na performance do sistema original. Isto é conseguido através da implementação de aspectos (semelhantes a classes Java) que introduzem o conceito de *advice* que descrevem o que fazer antes, depois ou em vez dos comportamentos endereçados por *joinpoints* e *pointcuts*, por exemplo invocações de determinados métodos em determinadas classes. Esses

aspectos são depois adicionados ao sistema em tempo de compilação através de técnicas de *weaving*. Embora na prática, este modo de instrumentação vá modificar o código, a maneira como o faz é transparente e controlada.

6.2.2 Modo de filtragem

O volume de informação extraído de uma captura tende a ser substancialmente grande e normalmente representa um *overhead* para a própria execução do sistema. Mesmo sistemas relativamente pequenos podem facilmente gerar milhares de objectos e milhões de invocações de métodos em pouco tempo de execução. Isto torna impraticável a análise da informação, mesmo com uma representação diagramática. Existem vários mecanismos para diminuir o volume de dados capturados na execução, uns mais vocacionados para a representação compactada e outros para a filtragem dos próprios eventos. Neste critério são caracterizados os mecanismos de filtragem de eventos que ajudam a capturar apenas a informação relevante. Numa escala ordinal, eles são representados como:

1. **Nenhum:** este nível descreve sistemas de capturas que não implementam qualquer mecanismo de filtragem de eventos.

2. **Ao nível de grafos de chamadas de modo estático:** existem sistemas de captura que recebem como argumento um conjunto de *call chains*. Um *call chain* é uma sequência de grafos de chamadas, ou seja, é uma sequência de extremidades E1,…,Ek do grafo em que o alvo de Ei é o mesmo que a fonte de Ei+1. Esta cadeia pode ser pensada como uma abstracção da pilha de execução de um sistema. Um grafo de chamadas é muito utilizado para representar relações de invocações entre métodos. Por exemplo, uma extremidade de um método m1 para um método m2 deve representar o facto de algum ponto dentro do método m1 ter potencialmente invocado o método m2. Os grafos de chamadas podem ser considerados como exequíveis ou não exequíveis, consoante exista ou não uma execução durante a qual haja correspondência com a configuração apresentada no grafo. A filtragem é realizada ao nível desses grafos de chamadas, ou seja, permite-se escolher quais os grafos de chamadas que se querem capturar, ignorando eventos fora desse âmbito.

3. **Ao nível de classes e métodos de modo estático, sem persistência de opções:** neste nível é possível escolher quais as classes e/ou métodos que se pretende filtrar/ignorar na captura. Esta escolha tem de ser feita antes da captura, de forma estática e cada alteração ou ajustamento exige uma nova captura. Os

dados a filtrar, as classes e métodos, podem ser obtidos de duas maneiras distintas: de diagramas de classes resultantes de *reverse engineering* estático do código fonte ou de dados recolhidos de capturas anteriores do sistema. Este nível é melhor que o anterior por permitir uma filtragem com maior nível de detalhe e por não necessitar de algoritmos de análise para construir os grafos de chamadas.

4. **Ao nível de classes e métodos de modo estático, com persistência de opções:** este nível vem estender as capacidades do nível anterior, na medida em que não necessita de várias capturas para representar alterações e ajustamentos nas opções de filtragem. Ou seja, os eventos devem ser todos recolhidos e guardados de modo persistente. Aquando da apresentação dos dados, apenas são recolhidos os que não foram filtrados. Alterações nas opções de filtragem, vão corresponder à recolha de diferentes dados, sem necessidade de voltar a capturar.

5. **Ao nível de classes e métodos em tempo de execução:** este nível descreve a possibilidade de, em tempo de execução, se poder interactivamente escolher que métodos ou classes se pretendem capturar. Essa escolha pode ser alterada e ajustada à medida que a execução do sistema decorre, permitindo visualizar os resultados das opções de filtragem tomadas de forma interactiva.

6.2.3 Tipo de captura

Uma captura de informação de um sistema pode ser feita de dois modos: estático ou dinâmico. Estes dois modos têm vantagens e desvantagens, e tendem a complementar-se em alguns aspectos. Para caracterizar este critério, foi definida a seguinte escala ordinal.

1. **Captura estática:** a captura estática, tal como o nome indica, retira informação do sistema de forma estática, ou seja, de um sistema sem estar em execução. Esta captura vai analisar o sistema representado pelo seu código fonte, retirando informações sobre a sua visão estática e representando-a comummente em diagramas de classes. No entanto, é também possível retirar, da análise do código, informação de modo a construir grafos de controlo de fluxo que estão na base da construção dos diagramas de sequência. Esta aproximação tem algumas vantagens no que respeita ao reconhecimento de primitivas de controlo, tais como ciclos ou execuções condicionais, mas não consegue conjugar a estrutura com a captura da forma como o sistema pode ser realmente utilizado, ou seja, os seus cenários de execução.

2. Captura dinâmica: o objectivo da análise dinâmica é recolher a sequência de interacção da execução real de sistemas. Ela deve ser capaz de apanhar os eventos que ocorrem durante a execução de um sistema, ou parte dele. A vantagem deste modo de captura é a recolha da informação que representa realmente o comportamento do sistema. Existem, no entanto, alguns condicionantes, como por exemplo a necessidade de ter um sistema (ou parte dele) executável ou a dificuldade em identificar primitivas de controlo, como condições e ciclos. Nesta abordagem, podemos também encontrar o problema de garantir a cobertura a 100%, se apenas se recorrer às informações capturas nas execuções, para realizar essa análise.

3. Captura com conjugação dos modos: a conjugação de ambas as capturas é outro mecanismo possível para facilitar a futura análise da informação recolhida. De facto, não é possível ter um elevado grau de confiança em análises de cobertura ou de consistência entre desenho e código, se esse julgamento for feito com base em diagramas construídos a partir de captura dinâmica. Por outro lado, os diagramas gerados a partir de análise estática, não permitem realizar estas análises ou tirar informações conclusivas. A junção da informação de ambos os métodos une o melhor dos dois mundos, resolvendo as faltas dos dois, permitindo analisar a cobertura e identificar os cenários de utilização do sistema.

6.2.4 Alvo de captura

A instrumentação e a captura podem ter diferentes tipos de alvo que podem restringir as opções de escolha dos mecanismos usados. Os dois alvos que são descritos são o código fonte e o código binário, segundo a escala nominal apresentada em baixo.

1. Código fonte: O principal problema é que os ficheiros com o código fonte podem não estar disponíveis. Por outro lado, permitir instrumentação destes ficheiros pode induzir erros imprevistos, alteração na performance, problemas de manutenção e ainda a confusão entre o código da aplicação e o código da instrumentação. No entanto, estes ficheiros são o alvo mais comum da instrumentação.

2. Código binário: corresponde a código compilado sob a forma de ficheiros *.jar* ou ficheiros *.class*. A possibilidade de usar código binário permite deixar o código fonte intocável. Isto previne a mistura entre o código da aplicação

e da instrumentação, e diminui a possibilidade de gerar erros extra. Por outro lado, para sistemas proprietarios, é possível que a empresa implementadora não se sinta à vontade em disponibilizar o código fonte para análise, principalmente se esta for feita em *outsourcing* (V&V independente). Este alvo torna alguns dos mecanismos de instrumentação, senão impossíveis, pouco viáveis, como é o caso da instrumentação manual.

6.2.5 Abrangência

A abrangência, neste estudo, prende-se principalmente com o uso de propriedades específicas, algumas descritas em versões do UML e outras próprias das linguagens de programação. Essas propriedades têm não só o problema da sua representação, mas também, e principalmente, a sua identificação na captura dinâmica de eventos. Para caracterizar os vários níveis de abrangência, é descrita de seguida uma escala ordinal (cumulativa).

1. **Estrutura do código:** a representação e identificação da estrutura do código é o primeiro nível da escala. A captura dinâmica dos eventos de interacção identifica, de forma clara, a visão comportamental do código implementado, ou seja, como o sistema se comporta durante a sua execução. Para além disso, é possível capturar informação relativa à parte estrutural do sistema, como as classes (objectos) que são chamadas, que métodos são invocados, estabelecendo assim as suas ligações. Um diagrama de sequência permite representar tudo isto, através das mensagens trocadas, da forma descrita na versão 1.4 do UML.

2. **Iterações e selecções:** este nível descreve a capacidade de capturar a existência de ciclos e intenções. Tanto a representação de iterações como de selecções, em diagramas de sequência, está prevista na versão 2.0 do UML. Na captura estática a identificação de ciclos e condições no código depende apenas da linguagem em estudo. No caso da captura dinâmica é necessário recorrer a técnicas de identificação de padrões e heurísticas, de modo a tentar identificar se uma certa sequência de eventos pode ou não ser considerada um ciclo de execução ou se um fluxo de execução é fruto de uma selecção.

3. **Paralelismo:** a abrangência do paralelismo significa a capacidade de identificar, e representar, eventos que traduzem situações de sistemas distribuídos e/ou sistemas com implementação de capacidades de paralelismo. Isto engloba as mensagens assíncronas provocadas por situações de *multithreading* local e ainda a

comunicação entre componentes que se encontram distribuídos. A representação de mensagens assíncronas está prevista quer na versão 1.4 quer na 2.0 do UML. No entanto, a identificação e ordenação (temporal) dessas mensagens é uma tarefa complexa e tem sido um tema em discussão em trabalhos semelhantes.

6.2.6 Armazenamento e representação dos dados

Para além dos mecanismos de captura e instrumentação do código, o modo como os dados recolhidos de cada captura são armazenados e representados dentro do sistema é muito importante. O armazenamento dos dados capturados, muitas vezes em grande quantidade, refere-se ao modo como estes ficam guardados, se com ou sem persistência. O modo de representação da informação é a maneira como, dentro das possibilidades de armazenamento, a informação é organizada e estruturada. Em primeira instância, a informação pode ser armazenada de modo não estruturado, mas podemos encontrar outras soluções como utilizar metamodelos existentes (UML, por exemplo) ou criar um metamodelo próprio (de raiz ou adaptando um existente). Para representar estes dois critérios, que se acredita indissociáveis, são estabelecidos dois indicadores que são conjugados, ou seja, os modos de armazenamento são representados com números e os modos de representação com letras. A conjugação número e/ou letra é a caracterização do trabalho em causa.

a. Sem persistência;	**a) Dados não estruturados;**
b. Ficheiros simples;	**b) Metamodelo existente;**
c. Base de dados;	**c) Metamodelo próprio.**

6.2.7 Exportação

A informação recolhida em capturas dinâmicas é representada a vários níveis, como o nível visual (representação gráfica) e o nível não visual (informação de apoio à construção dos modelos). Ambos os níveis podem ser exportados sob várias formas, consoante o propósito a que se destinam. No entanto, a exportação é sempre um meio eficaz que permite interoperabilidade entre ferramentas e flexibilização dos mecanismos de análise. Para este critério foi descrita uma escala ordinal que representa alguns dos mecanismos usados para exportação.

1. **Nenhuma:** este primeiro nível da escala traduz os casos em que não há qualquer tipo de exportação de informação prevista. Isto origina uma grande dependência à ferramenta que originou os resultados, ou pelo menos ao formato com que representa os dados (caso seja conhecido), impossibilitando a utilização dos resultados como ponto de partida de outras transformações e análises.

2. **Ficheiros de formato próprio:** esta exportação representa qualquer caso em que a exportação está prevista, mas é feita sob a forma de ficheiros com formato próprio (*custom)*. A utilidade desta exportação é bastante limitada, uma vez que não vem resolver nenhum dos problemas encontrados no nível anterior. Apenas pode ser usada como um mecanismo de persistência com formato proprietário para a própria ferramenta que a implementa.

3. **Ficheiro gráfico:** os ficheiros gráficos são úteis para casos em que a exportação de informação de modelos não é necessária, sendo a exportação da imagem suficiente. Existem inúmeras possibilidades de exportação de documentos gráficos. Duas dessas possibilidades são os ficheiros bitmaps e vectoriais.

4. **Ficheiros XML:** o uso de ficheiros XML pode ser útil para permitir um modo de serialização mais flexível e simples. Ele permite o uso de algumas transformações, nomeadamente através de ficheiros XSLT, para outros formatos, diminuindo a dependência a um tipo de solução. No entanto, este tipo de exportação continua a ser um problema no que respeita à interoperabilidade. O facto de o formato do ficheiro XML poder tomar qualquer forma, torna difícil, senão impossível, a partilha e circulação dos modelos.

5. **Ficheiros XMI:** XMI (XML Metadata Interchange) corresponde a uma norma definida pelo OMG, que permite representar e partilhar diagramas UML entre ferramentas de modelação e outros repositórios. A existência de um formato único para a representação dos modelos parece ser a solução mais eficaz para a interoperabilidade entre ferramentas e a flexibilização de resultados. No entanto, este processo ainda tem algum caminho a correr para a sua total implementação, na medida em que a maioria das ferramentas ainda não se dispôs a adoptar esta norma na sua completude, em parte explicado pela necessidade por

parte das produtoras de garantir uma fidelização (forçada) dos clientes aos seus produtos.

6.2.8 Modo de visualização

Um dos principais requisitos neste processo é a capacidade de reproduzir de algum modo a informação recolhida nas capturas efectuadas. No contexto deste estudo, o formato alvo de representação vão ser diagramas de sequência da linguagem UML. No entanto, existem várias abordagens possíveis para a representação dos diagramas, que diferem entre si em alguns aspectos, nomeadamente no modo como permitem a interacção com o utilizador. Embora a visualização dos diagramas não seja tratada nesta dissertação, foram descritos os vários modos de visualização segundo uma escala ordinal, para melhor contextualizar o problema.

1. **Ficheiro gráfico:** o modo mais simples de criar um formato visual dos diagramas é gerando formatos gráficos, nomeadamente ficheiros com o formato bitmap. Por ser uma representação estática, este tipo de formato não permite a edição dos diagramas e seus elementos.

2. **Ferramenta UML externa:** o uso de uma ferramenta externa que permita a visualização dos diagramas pode ser uma boa solução. O facto de se usar uma ferramenta já implementada e com todas as funcionalidades de uma ferramenta UML, permite não só simplificar e focar uma nova aplicação de captura em outras propriedades que não a visualização, mas também provoca uma menor alteração ao processo habitual dos utilizadores, que podem continuar a usar as suas ferramentas habituais de visualização de diagramas, com as quais já se sentem familiarizados. Por outro lado, a utilização de uma ferramenta externa exige mecanismos de exportação, de modo a que a ferramenta importadora reconheça a informação a representar.

3. **Animação:** a animação de diagramas vem tentar minimizar o problema de compreensão provocado pelas grandes proporções a que muitas vezes chegam esses mesmos diagramas. A animação representa o filme das interacções com uma visão temporal dos acontecimentos, tentando criar uma reprodução diagramática da execução do sistema. Formas de conseguir esta funcionalidade são o formato *bitmap* animado (GIF) e o vector gráfico SVG. A primeira possibilidade têm melhor suporte de ferramentas e visualização em browsers,

mas a opção do SVG permite obter melhor qualidade, escabilidade e o facto de serem ficheiros baseados em XML permite várias transformações até para outros formatos de representação.

1. **Visualização em tempo real:** a visualização em tempo real permite ao utilizador observar o comportamento de um sistema em tempo real durante a captura. Se for possível fazer uma filtragem em tempo real, este tipo de visualização passa a comportar-se como uma espécie de *debugger* visual. Este modo de visualização é tanto melhor conseguido se for realizado com recurso a animações, de preferência com possibilidade de edição. Entre as várias soluções possíveis para realizar esta funcionalidade destaco as soluções baseadas em SVG e as soluções de implementação de um *viewer costumizado*, que permitem maior flexibilidade e adequação ao problema.

6.2.9 Finalidade

A finalidade deste processo de captura origina diferentes opções de implementação que são importantes a ter em conta. Certas soluções são preferíveis consoante o alvo a que se destinam, por isso foi importante considerar este critério como caracterização dos vários trabalhos propostas na área. Para esquematizar a informação eles foram organizados segundo uma escala nominal.

- **Compreensão do código:** com o aumento da complexidade que os sistemas têm vindo a demonstrar, mecanismos que permitam a compreensão do código ou do comportamento dos sistemas tomam uma posição de grande importância. Esta compreensão pretende ter acesso à estrutura do sistema, a opções de implementação usadas e ao mecanismo de comunicação entre os vários componentes.

- **Manutenção do código:** a manutenção do código está prevista no ciclo de vida do software. No entanto, nem todos os sistemas conseguem ter documentação actualizada ao longo do seu desenvolvimento. Daí a necessidade, actualmente muito verificada, de ferramentas que consigam extrair informação que permita fazer essa manutenção. Ferramentas que recuperam a informação estrutural de um sistema, sob a forma de diagramas de classes, são hoje bastante usuais, mas a recuperação da informação comportamental dos sistemas não é tão comum e pode ser conseguida através da geração dos diagramas de sequência.

- **Testes de cobertura:** outra finalidade para mecanismos de captura e geração de diagramas de sequência é suportar a actividade de testes. Com as informações possíveis de serem recolhidas da captura de um sistema é possível realizar testes de cobertura, de verificação de implementação de requisitos, de qualidade das opções de implementação face ao desenho, etc.

6.2.10 Granularidade

A granularidade da captura e da representação é também um aspecto importante. Algumas opções de granularidade permitem uma representação simplificada dos diagramas, não demonstrando a totalidade dos eventos capturados ou compactando informação que não permite tirar algumas informações importantes. Claro que a finalidade destes mecanismos também está fortemente ligada à granularidade necessária. Para caracterizar alguns dos níveis de granularidade possíveis, é descrita uma escala ordinal em baixo, em que o primeiro nível corresponde ao menor nível de granularidade e o ultimo nível ao maior.

1. **Método:** este nível de granularidade representa as capturas que são feitas ao nível de um método, ou seja, capturam os eventos gerados dentro da execução de um método. Isto permite descrever a implementação do método em estudo, mas não permite ter uma visão global do sistema ou da árvore de execução. Dada a localização da captura, este nível de granularidade permite ser efectuado em sistemas completos ou em apenas partes deles.

2. **Classe-classe:** neste segundo nível de granularidade, os eventos são capturados, mas a identificação dos intervenientes é feita ao nível das classes. Isto significa que qualquer invocação feita por qualquer instância de uma dada classe vai ser representada pela mesma, sem distinção de objectos. O problema desta representação é que não permite representar instâncias de uma classe que apresentem comportamentos diferentes, embora já permita ter uma visão mais global do sistema.

3. **Objecto-Objecto:** o nível de granularidade objecto-objecto permite representar os eventos invocados pelos diferentes objectos no sistema, assim como identificar as criações e comportamentos dos objectos individualmente. A visão conseguida, para além de global, é detalhada e representa a execução real de sistemas orientados a objectos.

6.3 Trabalho Relacionado no âmbito de Captura de Diagramas de Sequência

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
Merdes2006	**2,3,4**	**3**	**3**	**1,2**	**1**	**1, b e c**	**1 a 5**	**4**	**3**	**3**

Experiences with the Development of a Reverse Engineering Tool for UML Sequence Diagrams: A Case Study in Modern Java Development [Merdes and Dorsch, 2006]

Objectivo: Este documento pretende reportar as experiências levadas a cabo pelos autores na construção de uma ferramenta de recuperação e visualização de diagramas de sequência. É descrito o estudo tecnológico das opções para as preocupações mais relevantes, como a recolha e representação dos dados, a visualização, a edição e as facilidades de exportação.

Resumo Técnico: O desenvolvimento de uma ferramenta que suporta a reconstrução do comportamento de um sistema em execução, tem de ter em conta a recolha dos dados, a sua representação num metamodelo adequado, a exportação da informação do metamodelo ou da representação gráfica, assim como o seu pós processamento e a sua visualização. Ao longo do documento são descritas várias opções para resolver cada uma das problemáticas atrás identificadas, sendo que no final de cada apresentação, é descrita a escolhida pelos autores e a correspondente fundamentação. Como exemplo, os autores apresentam várias técnicas possíveis para proceder à recolha dos dados na execução de um sistema, como a instrumentação manual do código, a instrumentação do ambiente de execução, o uso de uma *Java Debug Interface* ou o uso da tecnologia dos Aspectos para realizar o *weaving* dos mesmos com o código. Após descritas as vantagens e desvantagens de cada opção, os autores optaram por escolher a última como a mais indicada para o efeito.

Crítica: Este documento sintetiza várias aproximações possíveis à resolução das problemáticas inerentes à construção de uma ferramenta do género aqui descrito, identificando quais as suas maiores desvantagens e vantagens. No entanto, a maneira como o faz é um tanto superficial, na medida em que não explica nenhuma das técnicas, apenas as caracteriza.

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
Taniguchi2005	**Nd**	**1**	**2**	**1**	**2**	**1, a (àrvore)**	**Nd**	**Nd**	**1**	**2**

Extracting Sequence Diagram from Execution Trace of Java Program [Taniguchi, Ishio et al., 2005]

Objectivo: Neste documento, os autores propõem um método para extrair diagramas de sequência compactos através de informação dinâmica de sistemas orientados a objectos. São ainda descritas quatro regras que permitem realizar essa compactação dos diagramas.

Resumo Técnico: A execução de programas tende a gerar muita informação que necessita de ser reduzida tanto quanto possível. O método proposto, neste documento, compacta partes repetidas da execução abstraindo padrões de repetição e chamadas recursivas. Para representar a execução, os autores usaram uma árvore de invocações em que cada nó representa um método que foi invocado, com a sua assinatura e o ID do objecto em que foi invocado. De seguida, o documento descreve quatro regras usadas para identificar os padrões de repetição. A primeira regra detecta a repetição exacta da estrutura de invocação de métodos, e compacta-a. A segunda regra detecta a repetição da estrutura de invocações, mas cujos ID dos objectos possam ser diferentes, ou seja, esta regra não compara os identificadores dos objectos. A terceira regra detecta a repetição da estrutura de invocações, mas admite que algumas das invocações possam estar em falta. Por último, a quarta regra detecta invocações de métodos recursivos, ou seja, considera invocações do mesmo método em diferentes objectos como sendo recursivas. As estruturas compactadas são depois transformadas

em diagramas de sequência. Para o conseguir, os autores adicionaram anotações aos diagramas originais, mostrando quantas vezes um determinado método foi invocado, adicionando a cada objecto os identificadores de todos os objectos participantes e substituindo o bloco das chamadas recursivas por uma caixa indicadora.

Crítica: O primeiro problema que encontrei neste documento foi a falta da explicação de como o processo de captura da execução dos sistemas é feita. Por outro lado, acho que deveria ser mais explícito o método como fazem as verificações de aplicação das regras. A compactação dos diagramas, e correspondente aplicação das regras propostas, originam perdas de precisão na representação da execução real, porque assumem realidades que podem desviar ou modificar o comportamento real do sistema. Outro ponto a reparar são as anotações que se criaram para representar as várias regras nos diagramas que diferem das propostas pela UML. Isto traz problemas de compreensão e interoperabilidade entre modelos e ferramentas.

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
Rountev2005	**Na**	**1**	**1**	**2**	**1**	**1 (CFG)**	**1,00**	**Nd**	**3**	**1**

Static Control-Flow Analysis for Reverse Engineering of UML Sequence Diagrams [Rountev, Volgin et al., 2005]

Objectivo: O objectivo deste documento é encontrar mecanismos de representação de controlo de fluxos nos diagramas de sequência UML. São propostas extensões simples ao UML e é descrito um algoritmo de mapeamento do grafo de controlo para os diagramas.

Resumo Técnico: Em primeira instância os autores acreditam que ainda não é possível representar o controlo de fluxo (comportamento iterativo e selectivo) da execução dos sistemas com as primitivas disponibilizadas pela UML 2.0. Para solucionar este problema eles identificaram duas extensões simples ao UML, de modo a representarem o grafo de controlo de fluxo (*Control-Flow Graph –CFG*). O processo começa pela selecção de um método de uma das classes do sistema e depois gera o diagrama de sequência que representa as interacções accionadas pela invocação do método. Para cada um, separadamente, é analisado o controlo de fluxo presente no CFG e é criada uma estrutura de dados ao nível do método que guarda os aspectos mais relevantes do comportamento do mesmo. São identificados quatro controlos de fluxo que são particularmente importantes no âmbito do documento: opções, alternativas, ciclos e interrupções (*breaks*). De seguida, é descrito o processo de representação, nos diagramas propriamente ditos, dos passos anteriores, nomeadamente dos resultados da análise de cada CFG. São ainda apresentadas algumas discussões sobre decisões que os construtores de ferramentas deste contexto devem tomar, como por exemplo, a precisão vs a interoperabilidade, descrevendo o problema de se adicionar extensões ao UML para aumentar a precisão da informação

a representar, inviabilizando a interoperabilidade entre ferramentas, ou o tamanho vs a precisão.

Crítica: A análise estática do código, para representar o comportamento de sistemas é muito limitada. Devido a heranças, polimorfismo, ligações dinâmicas, etc, é muito difícil, senão impossível, saber que métodos (ou instruções no seu corpo) vão ser executados. Se juntarmos ainda sistemas com *multithreading* ou sistemas distribuídos o problema ainda piora mais. A solução aqui apresentada de apenas representar o possível comportamento de métodos é pouco útil para entender um sistema como um todo, ou para compreender o verdadeiro comportamento da sua execução.

Exploiting UML dynamic object modelling for the visualization of C++ programs [Malloy and Power, 2005]

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
Malloy2005	**4**	**4**	**3**	**1**	**1**	**Nd**	**Nd**	**4**	**3**	**3**

Objectivo: Neste documento é apresentada uma abordagem para modelar e visualizar as interacções dinâmicas entre objectos em aplicações C++. Os autores exploram os diagramas e grafos UML, tipicamente usados nas fases de identificação de requisitos e desenho, para visualizar as propriedades estáticas e dinâmicas de uma aplicação.

Resumo Técnico: C++ é mais difícil de analisar e não dispõe de um ambiente de execução rico e robusto quando comparado com a *Java Virtual Machine*. Para fazer face a isto, os autores decidiram usar a tecnologia do AspectC++, para permitir ao utilizador seleccionar as classes e métodos a serem analisados e depois realizar o *weaving* com a aplicação. Isto permite recuperar a informação necessária das partes seleccionadas, para gerar e visualizar os diagramas de sequência e colaboração

correspondentes. O processo proposto começa por tirar partido de forma estática de um diagrama de classes e de um grafo de invocações, para permitir seleccionar as partes da aplicação a modelar. São usados aspectos para posteriormente permitir a análise e visualização das partes seleccionadas. Para além disso, a selecção estática é complementada por selectores dinâmicos que permitem ao utilizador filtrar objectos e métodos em tempo de execução. O primeiro *output* da ferramenta é um diagrama de classes simplificado, que apenas apresenta as ligações de herança e de invocações. O segundo é o grafo de invocações do sistema. Depois a aplicação é executada e a ferramenta permite "andar pela execução" segundo passos, mostrando a informação recolhida sob a forma de diagramas de sequência e de colaboração em tempo de execução

Crítica: A abordagem proposta vem adicionar algumas particularidades de visualização interessantes que podem facilitar a análise do utilizador. No entanto, não sei até que ponto é que o sistema de visualização em tempo real, ou o modo de filtragem, não se tornam demasiado pesados, prejudicando a performance do sistema original, principalmente se for um sistema grande e/ou complexo. Por outro lado, a execução por etapas, para além de muito intrusiva, nem sempre permite observar o comportamento real. Se estivermos a analisar sistemas onde o tempo de resposta é muito importante, não seria possível utilizar a abordagem proposta.

Static and Dynamic Analysis of Call Chains in Java [Rountev, Kagan et al., 2004]

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
Rountev2004	**1**	**2**	**3**	**1**	**1**	**1 (Call Graph)**	**Nd**	**Nd**	**3**	**2**

Objectivo: Este documento procura apresentar uma framework para análise estática e dinâmica de cadeias de invocações em componentes Java. É também descrita uma ferramenta que realiza testes de cobertura.

Resumo Técnico: É proposto um framework para realizar a análise estática e dinâmica das cadeias de invocações em Java. Este baseia-se numa estrutura de dados

que é uma generalização de uma *calling context tree* (árvore do contexto de invocações). O processo apresentado considera a análise de cadeias de invocações que recebe como *input* um grafo de invocações criado por um algoritmo externo. Os resultados devem ser apresentados de maneira compacta e devem ser fáceis de usar na análise dinâmica. Por sua vez, a análise dinâmica deve receber um conjunto de cadeias de invocações criadas pela análise estática, e deve reportar a cobertura de execução desses cadeias. O modo usado, para tirar informações da execução que permitam analisar a cobertura, é a instrumentação manual do código fonte, à custa de *triggers* que capturam certos eventos. Os autores descrevem ainda um conjunto de experiências que avaliam a imprecisão das instâncias da framework.

Crítica: Embora esta abordagem conjugue a análise estática com a análise dinâmica de sistemas, volta estar presente a limitação de compreensão de um sistema como um todo (desta vez, a granularidade é ao nível do componente, classe, em vez do método, como vimos anteriormente). Por outro lado, a finalidade deste documento não é recuperar o modelo do comportamento do sistema em primeira instância, mas mais especificamente efectuar análise de cobertura e de viabilidade de cadeias de invocações. O modo usado para instrumentação do código, apesar de apenas ser feito para os componentes de interesse em cada execução, é demasiado intrusivo, propenso a erros e consome tempo e esforço, desnecessário se tivesse sido escolhido um mecanismo mais automatizado.

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
Briand2003	**4**	**1**	**2**	**1,2**	**3**	**Nd, c**	**Nd**	**Nd**	**3**	**3**

Towards the Reverse Engineering of UML Sequence Diagrams for Distributed, Multithreaded Java software [Briand, Labiche et al., 2003]

Objectivo: Este documento propõe uma metodologia e infra-estrutura de instrumentação para fazer *reverse engineering* de diagramas de sequência UML a partir da análise dinâmica. O intuito é, não só apoiar a compreensão do

comportamento de sistemas legados, mas também de suportar a verificação e validação da qualidade na implementação dos requisitos.

Resumo Técnico: Em qualquer abordagem com o intuito de fazer *reverse engineering* de diagramas de sequência UML são necessárias solucionar três questões importantes: o modo de instrumentação do código, a estratégia de representação da informação e a fusão dos cenários num só caso de utilização. No entanto, apenas as duas primeiras são discutidas neste documento. A instrumentação do código traz o dilema de se guardar a versão "limpa" código, a versão instrumentada ou ambas. Para minimizar este problema, os autores decidiram usar uma estratégia menos intrusiva, a programação orientada a aspectos. Outro desafio encontrado pelos autores foi a representação e identificação de primitivas de controlo (selecções e ciclos) e de características particulares de sistemas *multithreading* e distribuídos. De modo a formalizar a sua abordagem, foram especificados dois metamodelos: um para as execuções e outro para os diagramas de sequência, com regras de mapeamento entre eles definidas usando OCL (*Object Constraint Language*).

Crítica: Este documento traz uma descrição técnica sobre a implementação de mecanismos de *reverse engineering* para diagramas de sequência. Embora seja muito exaustivo na demonstração de decisões de implementação, existem alguns pontos que não são focados, como a visualização dos diagramas gerados. O tratamento de casos de sistemas distribuídos e *multithreading* é sem dúvida uma ajuda para todos os autores que estão a fazer trabalho semelhante, no entanto existem algumas decisões tomadas na proposta que identifico como desvantagens. A primeira é o uso de primitivas de controlo para representar comportamentos de ciclos ou selecções. Esta decisão é útil para compactar o diagrama gerado e fornecer uma visão sintetizada do comportamento do sistema, mas há situações em que a análise dos eventos tal como aconteceram pode ser importante e com as primitivas tal não é possível. Uma solução poderia ser permitir ao utilizador escolher que modo de representação que seria mais favorável. Por outro lado, o objectivo de criar um só diagrama de sequência para demonstrar todos os comportamentos possíveis pode poluir o diagrama e tornar difícil a sua compreensão e análise, principalmente para sistemas com cenários grandes e/ou complexos.

6.4 Resumo da Taxionomia

Propostas	Merdes2006	Taniguchi2005	Rountev2005	Malloy2005	Rountev2004	Briand2003
Mecanismo de instrumentação	**2,3,4**	**Nd**	**Na**	**4**	**1**	**4**
Modo de Filtragem	**3**	**1**	**1**	**4**	**2**	**1**
Tipo de Captura	**3**	**2**	**1**	**3**	**3**	**2**
Alvo de Captura	**1,2**	**1**	**1**	**1**	**1**	**1,2**
Abrangência	**1**	**2**	**1**	**1**	**1**	**3**
Armazenamento e representação dos dados	**1, b e c**	**1, a (àrvore)**	**1 (CFG)**	**Nd**	**1 (Call Graph)**	**Nd,c**
Exportação	**1 a 5**	**Nd**	**1**	**Nd**	**Nd**	**Nd**
Modo de visualização	**4**	**Nd**	**Nd**	**4**	**Nd**	**Nd**
Finalidade	**3**	**1**	**3**	**3**	**3**	**3**
Granularidade	**3**	**3**	**1**	**3**	**2**	**3**

Por todos os documentos analisados, foi possível encontrar abordagens inovadoras e importantes para o desenvolvimento de uma ferramenta, que tem como objectivo implementar a funcionalidade de *reverse engineering* de diagramas, que representem o comportamento de sistemas. Apesar disso, não existe nenhum documento que descreva todas as técnicas identificadas como necessárias, em conjunto e de forma eficiente. Uns autores focam mais características de visualização, outros de reconhecimento de primitivas de controlo e ainda outros que descrevem de forma exaustiva a implementação do seu mecanismo. Um dos documentos vêm adicionar informação crucial à resolução de problemas como o *multithreading,* enquanto que outros se distanciam bastante do objectivo desta dissertação. Muitas decisões tomadas vão limitar posteriormente, quer a visualização dos resultados quer a sua análise, pelo que é importante a identificação clara do contexto para que cada documento foi idealizado.

Pelas propostas apresentadas observamos que, recentemente, se têm vindo a procurar e a usar técnicas de instrumentação de código que sejam o mais “leves” para a aplicação original, causando menos danos de performance possíveis e diminuindo o

nível de intrusão. A técnica escolhida pela maioria é o *weaving* dos aspectos com o código. Esta técnica, não só as vantagens já referidas, como fornece um meio de filtragem bastante eficiente e detalhado.

Existe ainda uma lacuna, muitas vezes observada, no mecanismo utilizado pelos autores para armazenar os dados recolhidos, representá-los ou mesmo exportá-los. Apenas um autor descreveu a grande importância da exportação da informação, referindo a possibilidade de usar o XMI para garantir a interoperabilidade entre ferramentas.

6.5 Framework para a caracterização dos cartões CRC

Na análise do trabalho relacionado sobre cartões CRC foram identificados alguns critérios específicos que serviram de base de comparação entre as várias propostas. Esses critérios tiveram em consideração não só o modo de construção dos Cartões CRC, mas também o modo como estes se apresentam e o contexto em se inserem. Os níveis de cada critério são apresentados segundo escalas ordinais.

- **Fase de desenvolvimento:** Estas fases de desenvolvimento estão relacionadas com as fases previstas no ciclo de vida do software: análise, desenho, implementação, testes, manutenção [Silva and Videira, 2005]. O aumento da complexidade dos sistemas actuais com as constantes alterações dos requisitos, a rotatividade das equipas de desenvolvimento e a pressão exercida nas mesmas para produzirem *outputs* dos sistemas em curtos espaços de tempo, levam a que a documentação existente (quando existe) sobre o sistema, não seja actualizada ao longo da fase de construção, criando o que se conhece por sistemas legados. Existe por isso uma grande preocupação em criar mecanismos que consigam retirar informações dos sistemas de modo a gerar essa documentação indispensável. Esses mecanismos são conhecidos como mecanismos de *Reverse Engineering* [Pressman, 2005]. As fases que têm representatividade neste estudo estão agrupadas segundo a seguinte escala ordinal:

    1. **Análise precoce:** por análise precoce entenda-se o início da fase de análise. As actividades levadas a cabo nesta altura são essencialmente actividades de levantamento de requisitos, nomeadamente definição e identificação dos requisitos do sistema como um todo, em que existe ainda um elevado grau de abstracção no que respeita à percepção da estrutura do sistema;

2. **Análise:** esta fase difere da fase anterior por apresentar um grau de detalhe superior. A preocupação principal nesta altura é a identificação mais detalhada dos requisitos para o bom funcionamento do sistema, tendo já conhecimento adquirido sobre o contexto e principais funcionalidades que devem estar presentes. São construídos modelos conceptuais que auxiliam o entendimento do problema, sem iniciar o desenho da solução;

3. **Desenho:** na fase de desenho, são identificados e construídos os modelos que devem representar a solução final. É definida a arquitectura do sistema, assim como dos componentes, interfaces, e outras características com um menor grau de abstracção. Deve ser descrito com detalhe como o software vai ser decomposto e organizado em componentes, e como esses se devem comportar.

4. **Fases tardias:** incluem as fases de construção, testes e manutenção. Podemos encontrar actualmente vários mecanismos de apoio à recuperação de documentação de sistemas (mecanismos de *Reverse Engineering*), sendo nas fases mais "tardias" do ciclo de vida que esses mecanismos têm o seu foco de utilização.

- **Geração:** Este critério descreve o modo como se obtêm os Cartões CRC. Dentro das várias possibilidades foi estabelecida a seguinte escala ordinal.

    1. **Geração manual:** a geração manual descreve o processo de identificação dos constituintes de cada cartão e sua posterior criação. Isto deveria ser realizado no do processo de análise de requisitos, individualmente pelo analista ou em grupo, juntando *stakeholders* e outros intervenientes que pudessem ajudar a entender a essência do sistema.

    2. **O uso de um editor genérico sem semântica específica:** para transformar os cartões físicos em formato electrónico, podem ser usadas várias aplicações. A mais simples e também menos adequada, é usar um editor genérico, como o *Notepad* ou outro similar, que não oferece qualquer funcionalidade de apoio específico à criação dos cartões. Embora esta solução seja já uma evolução ao nível anterior, é bastante limitada e acaba por representar um esforço adicional na construção

electrónica dos cartões por não oferecer nenhum mecanismo de suporte. Outra desvantagem das abordagens electrónicas em geral é não permitir a "encenação da construção" dos cartões na definição e verificação das responsabilidades.

3. **O uso de um editor dedicado com semântica proposta ou usando semântica de modelação:** usar um editor específico para a construção dos cartões CRC vem minimizar algumas das limitações existentes no nível anterior. Alguns editores permitem a construção dos cartões em simultâneo com actividades de modelação, como criação de diagramas de sequência, etc, oferecendo uma espécie de *round-trip engineering* entre os cartões e os modelos. Outros disponibilizam uma semântica dedicada a cartões CRC, permitindo adequar a aplicação e a sua utilização ao contexto em causa.

4. **O uso de aplicação com funcionalidade de *reverse engineering* específico para o efeito:** o nível superior da geração de cartões CRC representa aplicações que conseguem, através de mecanismos de *reverse engineering*, extrair informações do sistema que permitem a geração automática dos mesmos. Este tipo de aplicações diminui o *overhead* necessário à criação dos cartões. No entanto, devido à sua natureza (por necessitarem de um sistema construído) apenas podem ser usadas em fases tardias do ciclo de vida do software para documentação, manutenção e teste dos mesmos.

- **Origem:** este critério descreve o contexto da criação e o modo de acompanhamento da construção de cartões CRC, ou seja como essas actividades são levadas a cabo. O modo originalmente proposto foi adequar a construção dos cartões com as outras actividades usuais do levantamento de requisitos. Entre eles está a inspecção visual de informação e as reuniões, mais ou menos participadas, com os *stakeholders*. A partir desta abordagem mais arcaica surgiram outras que têm vindo a apresentar mais autonomia, tentando diminuir o tempo e o esforço de produção, não só dos cartões mas também das restantes actividades do ciclo de vida. Nestas abordagens, mais automáticas, tem-se o *parsing* de texto ou a recolha automática de informação. Em formato de escala

ordinal, é apresentada em baixo uma breve descrição de cada uma das técnicas consideradas neste critério.

1. **Inspecção visual:** a inspecção visual da especificação dos requisitos é a actividade de análise de documentos, procurando extrair as informações necessárias à identificação das partes constituintes dos cartões: as classes, as responsabilidades e as colaborações. Esta actividade pode ser acompanhada de reuniões com os *stakeholders*, em que estes podem ter uma atitude mais ou menos participativa na construção dos cartões.

2. **Parsing de texto:** o *parsing* de texto é outra forma existente para a interpretação dos dados importantes à construção dos cartões. Este método tira partido de aplicações que efectuam o reconhecimento de verbos e nomes em documentos de texto, identificando os potenciais elementos e requisitos, e permitindo a posterior validação do utilizador.

3. **Identificação automática a partir de um sistema existente**: Por último, é ainda considerada a origem automática das informações necessárias à construção dos cartões CRC, ou seja, a identificação automática dos cartões a partir de sistemas existentes. Embora este modo de criação de cartões CRC seja o mais prático e fidedigno, tem a desvantagem de apenas ser possível de usar nas fases tardias de desenvolvimento, por exigir um sistema construído (ou parcialmente construído).

- **Granularidade:** A granularidade descreve o nível a que se descrevem e identificam as responsabilidades dos cartões CRC. Está associada ao nível de abstracção com que se criam e analisam os cartões. Em baixo, foram definidos três níveis de granularidade, apresentados segundo uma escala ordinal.

  1. **Responsabilidades representam casos de utilização (*use cases)*:** Este nível de granularidade, por ter um elevado nível de abstracção, apenas permite ter uma visão muito global das relações e do comportamento dos constituintes do sistema. Se um caso de utilização tiver (como é comum) vários cenários, não é possível identificar como se processa o comportamento da classe perante as variações dos cenários.

2. **Responsabilidades representam cenários de utilização:** Esta abordagem, inicialmente proposta, permite ter a noção de como cada classe colabora e está envolvida na execução de cada cenário, distinguindo as várias variações que podem ocorrer na execução do sistema.

3. **Responsabilidades representam métodos de classes:** Esta abordagem é excessiva no detalhe da informação que pretende representar. Além disso, um método pode ser invocado em diversas situações no decorrer da execução de um sistema e tomar comportamentos distintos. Perde-se a informação de que situações originam determinado comportamento, porque o nível de abstracção é demasiado baixo e tenta apenas mostrar o nível de implementação do método. Outra desvantagem é o esforço e o tempo extra necessário à identificação deste detalhe de informação.

6.6 Trabalho Relacionado no âmbito de Cartões CRC

CRC Cards for product modelling [Hvam, Jesper et al., 2003]

	Fase de desenvolvimento	Geração	Origem	Granularidade
Hvam2003	**2**	**Nd**	**1**	**2**

Objectivo: Face à necessidade encontrada de desenvolver um processo genérico e técnicas de modelação associadas à construção de modelos de produtos, os autores apresentam o uso de cartões CRC adaptados como possível solução para o efeito.

Resumo Técnico: Este documento propõe um processo que guie a construção e modelação de produtos. De entre os processos existentes, os autores seleccionaram e adaptaram o ciclo de vida do projecto ICAM (*Integrated Computer Aided Manufacturing*) descrevendo as várias fases que o constituem e explicando o processo que deve ser acompanhado. A construção dos cartões CRC é proposta acontecer na fase de análise, mais precisamente na análise orientada a objectos. No entanto, os autores viram a necessidade de adaptar os cartões tradicionais ao contexto de modelação de produtos. Para isso, adicionaram novos campos como as *subclass/superclass* e *subparts/superparts* que correspondem a ligações de

generalização e a agregação entre classes, *sketch* que corresponde ao esboço do produto, *know/Does* que descreve o que a classe deve conhecer e deve fazer.

Crítica: O âmbito de utilização destes cartões desvia-se do que se pretende nesta dissertação. Embora o problema seja a origem de sistemas legados, este documento foca-se apenas na problemática da modelação de produtos. O procedimento para a construção e modelação dos produtos também não utiliza o UML, o que poderá originar problemas como a origem de múltiplas técnicas sem nenhuma informação padrão que as ligue. A criação de alguns dos campos dos cartões torna-os redundantes com os já existentes, como por exemplo as responsabilidades e os novos campos *know/does*. Por outro lado, existem outros campos, como as agregações e generalizações, que vão além do que se espera para um cartão, diminuindo uma das grandes vantagens desta abordagem que é precisamente a simplicidade. O modo de geração não é explícito, sendo apenas referido que seria útil ter um formato electrónico para os cartões que permitisse a rápida identificação do impacto de alterações, ou mesmo que automaticamente os actualizasse em resultado disso.

The EasyCRC Tool [Raman and Tyszberowicz, 2007]

	Fase de desenvolvimento	**Geração**	**Origem**	**Granularidade**
Raman2007	**1,2**	**3**	**2**	**2**

Objectivo: Para mitigar o problema da falta de ligação entre a técnica dos cartões e dos diagramas de sequência, os autores propõem uma ferramenta, denominada de *EasyCRC*, que combina as duas.

Resumo Técnico: Neste documento é proposta uma ferramenta que vem automatizar o processo de definição de classes, responsabilidades e colaborações. O processo começa com a disponibilização de documentos de requisitos de um sistema, escritos em língua natural. A ferramenta proposta deve então proceder à sua análise de modo a identificar nomes e verbos no texto que possam vir ser identificadas como classes. Cabe ao utilizador validar as opções propostas, originando a criação automática dos respectivos cartões vazios. A segunda funcionalidade desta ferramenta é identificar as responsabilidades e as colaborações para cada cartão. Para isso, ela permite a construção de diagramas de sequência, em que para cada mensagem criada no diagrama é possível adicioná-la como responsabilidade no respectivo cartão. No fim,

a ferramenta permite exportar os resultados em formato XMI (XML Metadata Interchange) ou HTML.

Crítica: Esta ferramenta vem minimizar o problema da complexidade de grandes sistemas, que se tornava inviável com a técnica tradicional dos cartões CRC. Tem também mecanismos positivos que permitem tornar o processo de criação mais automático. No entanto, traz duas desvantagens. A primeira prende-se exactamente com o modo de criação. A forma automatizada com que é feito o *parsing* dos ficheiros para identificar classes, deixa de necessitar da presença de outros intervenientes e do seu conhecimento. Isto pode ser bom, no sentido de diminuir o tempo e recursos necessários, mas afasta-os do processo de modelação criando um fosso entre eles e o sistema. Por outro lado, o facto de ser necessário a construção dos diagramas de sequência para a construção dos cartões, pode ser desmotivante. Em muitos casos, os diagramas de sequência não são criados por ser difícil fazê-lo e principalmente mantê-los. Isto pode originar que se deixe também de fazer os cartões CRC ou que se abandone o uso da ferramenta.

CRC Modelling: Bridging the Communication Gap Between Developers and Users [Ambler, 1998]

	Fase de desenvolvimento	**Geração**	**Origem**	**Granularidade**
Ambler1998	**1**	**1**	**1**	**2**

Objectivo: O objectivo deste artigo é apresentar um processo para a modelação dos cartões CRC, identificando claramente seis passos a seguir para a sua correcta concretização, com o intuito de envolver de igual modo quem desenvolve e quem usa o sistema.

Resumo Técnico: O primeiro passo proposto, "*Put together the CRC modelling team*", descreve quem deve estar presente numa sessão de modelação CRC e identifica claramente que papéis devem assumir cada um dos presentes. O segundo passo, "*Organize the room"*, explica o que deve existir na sala onde se processa a sessão. O terceiro, "*Brainstorm*", identifica técnicas de geração de ideias, como questões que devem ser feitas, para facilitarem o processo de identificação e compreensão dos requisitos. O quarto passo, "*Explain the CRC Modelling technique*", serve para relembrar que é necessário explicar o que é e como se processa a modelação de CRC, antes de dar início a uma sessão. O quinto passo, "*Iteratively*

perform the steps of CRC modelling", descreve os passos previstos na modelação de CRC: identificação das classes e das responsabilidades, definição das colaborações e dos casos de utilização e organização espacial dos cartões numa mesa. Por fim, o sexto passo, "*Perform use-case scenario testing*", explica a tarefa de validação e verificação dos requisitos dos utilizadores. A ideia é descrever cada cenário de utilização e verificar se o modelo CRC reflecte os requisitos necessários.

Crítica: O processo descrito necessita da comparência de um conjunto de pessoas cujos horários são muitas vezes difíceis de conjugar. Para além disso, os utilizadores podem sentir-se intimidados por discutir, com quem vai desenvolver o sistema, certas questões que podem ser de carácter mais técnico, dificultando a comunicação entre ambos. Se o sistema a desenvolver for um sistema de grandes dimensões, esta técnica pode apresentar-se inviável e bastante confusa, face ao número de pessoas envolvidas. A verdade é que podem surgir diferentes visões dos problemas prolongando todo o processo e consumindo tempo que nem sempre é possível dispender.

Essential Use Cases and Responsibility in Object-Oriented Development [Biddle, Noble et al., 2002]

	Fase de desenvolvimento	**Geração**	**Origem**	**Granularidade**
Biddle2002a	**1**	**1**	**1**	**1**

Objectivo: O objectivo deste artigo é explorar a aplicação directa dos *essencial use cases* no desenvolvimento tradicional de software orientado a objectos. Pretende-se comprovar que estes casos de utilização funcionam tão bem como os tradicionais, no início do desenvolvimento, que suportam melhor a captura dos requisitos, graças à independência da tecnologia, e ainda que permitem suportar melhor a comunicação entre quem desenvolve e os restantes intervenientes (stakeholders).

Resumo Técnico: O documento começa por descrever as diferenças entre os casos de utilização tradicionais e os *essencial use cases*. A principal diferença é que os primeiros foram criados para suportar o desenho de software orientado a objectos, enquanto que os segundos foram introduzidos para o desenho e desenvolvimento de interfaces com os utilizadores. A questão que é posta aqui, é se os *essencials use cases* não poderão fazer parte do desenvolvimento de software orientado a objectos, aplicando as suas vantagens de simplicidade a esta abordagem. Os autores iniciaram

esta exploração, porque queriam melhorar o entendimento dos casos de utilização tradicionais e queriam tornar a sua elaboração como uma actividade de grupo. O que foi proposto foi adaptar a técnica dos Cartões CRC para a análise e avaliação formal dos *essencial use cases*. O processo começa com a identificação de *essencial use cases* para o sistema, preenchendo uma espécie de cartão CRC, em que em cima se coloca o nome do *use case*, do lado esquerdo as intenções dos utilizadores e do lado direito as responsabilidades do sistema, face às intenções descritas. A ligação com o desenvolvimento de software orientado a objectos faz-se na segunda parte do processo. Uma vez que nos primeiros cartões foram descritas responsabilidades do sistema, o segundo passo, descreve a criação classes, através dos cartões CRC tradicionais, e distribui essas responsabilidades a cada uma delas.

Crítica: O âmbito deste documento é um pouco desviado do apresentado nesta dissertação. A inovação à técnica tradicional dos cartões CRC foi principalmente o ponto de partida da sua criação. Por outro lado, limitar a identificação de responsabilidades do sistema a intenções dos utilizadores em relação a interfaces, pode não ser muito satisfatório, principalmente se o software a desenvolver não tiver necessidade de as ter. A adaptação da técnica dos cartões CRC para a descrição dos *essencial use cases* não me parece adequada, uma vez que o formato original dos mesmos prevê a identificação de responsabilidades e das colaborações de cada classe, o que não faz sentido no caso de casos de utilização. Para além disso, existem actualmente técnicas bastante satisfatórias para a identificação e descrição dos casos de utilização e seus cenários.

Improving CRC-Cards Role-Play with Role-Play Diagrams [Borstler, 2005]

	Fase de desenvolvimento	Geração	Origem	Granularidade
Borstler2005	**1,2**	**1**	**1**	**3 (mensagens)**

Objectivo: Os autores identificam alguns problemas na técnica dos cartões CRC, como a substituição dos objectos pelos cartões durante a identificação da execução dos cenários, e a dificuldade de documentar ou seguir a execução dos cenários directamente (“on the fly”). Face a estes problemas, os autores propõem adicionar à técnica tradicional de criação dos Cartões CRC, a construção de novos diagramas

que descrevem a execução de cada cenário, combinando os diagramas de objectos e os de colaboração do UML.

Resumo Técnico: Após encontrados os problemas, os autores estabelecem uma clara separação na técnica tradicional dos cartões CRC: a parte de desenvolvimento inicial do modelo CRC e a segunda em que se definem e executem os cenários para propósitos de validação. Em primeiro lugar, é descrita a confusão entre classes e objectos que a técnica tradicional dos cartões CRC origina. Na verdade, embora cada cartão se refira a uma classe, muitos autores, incluindo os originais, falam de objectos. Para solucionar este problema, os autores recomendam que não se fale em objectos numa fase inicial, mas apenas em classes candidatas. No entanto, propõem que mais tarde, na fase de validação, se criem *object cards* que deverão ser instâncias dos Cartões CRC criados. Para a parte da descrição dos cenários e validação do modelo CRC, os autores propõem um novo tipo de diagrama, designado de *Role-play diagram*. Este diagrama descreve a execução dos cenários, representando os objectos, que são *object cards*, as suas ligações, através de linhas entre os objectos, e os pedidos (mensagens) que correspondem a responsabilidades presentes nos cartões.

Crítica: Existem algumas inovações propostas neste documento. A primeira é a criação de *objects cards*, para representar um objecto real. Na minha opinião, cada cartão CRC deve incluir as responsabilidades de cada objecto, instância da classe que se está a representar. O modo como cada objecto se comporta é descrito através de vários diagramas, alguns deles presentes na UML. A segunda proposta é a criação e construção de diagramas descritores de cenários. Ora, mais uma vez, a UML dispõe de diagramas criados para o efeito, como por exemplo os diagramas de sequência, que também identificam os objectos e as mensagens trocadas entre eles. Penso que os diagramas propostos representam redundância de informação às técnicas e artefactos já existentes. Para além disso, para sistemas de grande dimensão, em particular com cenários complexos, estes diagramas propostos podem ser difíceis de construir, visualizar e manter.

Reflections on CRC Cards and OO Design [Biddle, Noble et al., 2002]

	Fase de desenvolvimento	**Geração**	**Origem**	**Granularidade**
Biddle2002b	**2,3**	**1**	**1**	**2**

Objectivo: O objectivo deste documento é relatar as experiências tidas pelos autores ao introduzirem a técnica dos cartões CRC nos seus cursos de orientação a objectos. Eles descrevem as vantagens e fraquezas encontradas nessa introdução, e dão alguns concelhos de como as fraquezas podem ser ultrapassadas.

Resumo Técnico: Embora os autores já tivessem incluído no seu ensino, a técnica dos cartões CRC há alguns anos, decidiram apresentar um relatório mais específico da sua utilização em cursos realizados recentemente. Este relatório identifica as vantagens e as fraquezas encontradas, propondo estratégias que minimizem as fraquezas. As vantagens encontradas para a técnica dos cartões CRC é que os cartões facilitam a discussão aberta da estrutura estática e dinâmica do sistema e disponibilizam heurísticas que guiam o desenho na identificação de classes, responsabilidades e colaborações. O primeiro fenómeno descrito pelos autores é a relutância de várias equipas de alunos em usar os cartões, numa fase inicial. Eles preferiam tomar notas e deixar a construção dos cartões apenas para representar o desenho final. Os autores concordam em que as notas não são prejudiciais desde que se mantenha a discussão justa. Outro problema encontrado foi a confusão dos alunos entre objectos e classes. A proposta aqui é para enfatizar que um cartão representa um objecto e não uma classe, e depois mais tarde mostrar que vários objectos podem ser generalizados por uma classe. Uma outra fraqueza encontrada foi a identificação dos casos de utilização que muitas vezes não tinham em conta cenários alternativos. A maneira encontrada pelos autores para minimizar esta falta foi fazer cada grupo discutir os seus resultados em frente dos restantes, permitindo que a visão dos colegas levantasse opções diferentes e criando uma maior pressão em identificar os vários casos possíveis. Por fim, os autores recomendam a filmagem de cada sessão para manter todas as informações e soluções alternativas guardadas para futuros visionamentos.

Crítica: Este documento refere-se principalmente à utilização da técnica dos cartões CRC no ensino e descreve a sua aceitação. O contexto desta dissertação não é a utilização dos cartões no ensino, nem tão pouco a verificação da utilização dos mesmos. Por outro lado, parece-me que a estratégia de considerar os cartões ao nível de objectos é um pouco excessiva, no sentido que cada utilizador da técnica deve ter a percepção de que uma classe deve ser instanciada em vários objectos, podendo estes ter comportamentos diferentes. Acho que esta simplificação poderá limitar a compreensão inicial desta característica da orientação por objectos.

A Pattern for an Effective Class Responsibility Collaborator (CRC) Cards [Fayad, Hamza et al., 2003]

	Fase de desenvolvimento	Geração	Origem	Granularidade
Fayad2003	**1**	**1**	**1**	**2**

Objectivo: Os autores deste artigo identificaram algumas desvantagens na técnica dos cartões CRC. O primeiro problema é a possibilidade de baixa coesão e elevado *coupling*, resultante da sobrecarga de responsabilidades para uma determinada classe. O segundo problema é a criação de *macho classes*, que são classes que fazem a maior parte do trabalho do sistema, deixando apenas situações menores para as restantes classes. O terceiro problema é a exclusão de serviços disponibilizados por uma classe, pois para limitar a duplicação das funcionalidades, classes com serviços similares são agrupadas. O quarto problema é a não definição clara do papel de cada classe, porque para além de levar à assumição de responsabilidades erradas, pode ignorar situações de generalização e especificação. O quinto problema é a dificuldade em definir as responsabilidades, que aliadas ao ponto anterior podem levar à identificação de responsabilidades ambíguas ou irrelevantes. Por ultimo, o sexto problema é a dificuldade em mapear as classes associadas aos cartões, a classes nos diagramas de classes e aos cenários dos casos de utilização, principalmente se as classes tiveram muitas responsabilidades associadas a elas. O objectivo deste documento é propor uma nova visão dos Cartões CRC que tenta evitar os problemas descritos.

Resumo Técnico: A maior vantagem desta técnica é a sua simples compreensão. No entanto, essa simplicidade pode não transmitir toda a informação necessária para os passos seguintes do desenvolvimento. Por exemplo, a secção das colaborações não tem informação do tipo de colaboração que existe, ou seja, o cartão não especifica o serviço que a classe oferece às outras classes com que colabora. Por outro lado, ter demasiada informação num cartão pode pôr em causa a sua utilização e entendimento. É também importante compreender o papel de cada classe no sistema para identificar responsabilidades. No entanto, uma classe pode ter diferentes papéis e não existe nos cartões nenhuma maneira de o representar. Para além disso, a identificação das responsabilidades é crucial para os restantes passos do desenvolvimento. Uma classe pode apresentar diferentes responsabilidades, mas a

sua identificação num único cartão pode ser confuso. Dito tudo isto, é fácil entender que a construção eficaz dos cartões não é fácil, por isso os autores propõem um novo olhar sobre os cartões criando uma versão aumentada dos mesmos, para tentar solucionar os problemas por encontrados. A nova versão dos cartões vai incluir claramente um papel para cada classe, de maneira a que uma classe apenas possa estar associada uma responsabilidade. Também a secção das colaborações foi acrescentada, subdividindo-se em *Servers* e *Clients*. Nos *servers* são colocados os métodos que permitem implementar uma responsabilidade, enquanto que nos *clients* ficam todas as classes que colaboram com a classe particular através do pedido de serviços.

Crítica: Embora alguns dos problemas encontrados sejam legítimos, na minha opinião a solução proposta traz ainda maiores desvantagens. A forma que foi encontrada para tentar solucionar o facto de uma classe poder ter vários papéis num sistema, não resolveu nada. Na verdade separar por várias classes o que deveria estar representado apenas numa, vem trazer maior complexidade à análise e compreensão dos cartões. Do mesmo modo, limitar cada a classe a ter apenas uma responsabilidade vai de encontro à natureza das classes e possíveis instâncias. A identificação das colaborações divididas por serviços e clientes, aumenta a dificuldade e traz um menor nível de abstracção à criação dos cartões.

6.7 Resultados da análise

Critérios	Fase de desenvolvimento	Geração	Origem	Granularidade
Hvam2003	**2**	**Nd**	**1**	**2**
Raman2007	**1,2**	**3**	**2**	**2**
Ambler1998	**1**	**1**	**1**	**2**
Biddle2002a	**1**	**1**	**1**	**1**
Borstler2005	**1,2**	**1**	**1**	**3 (mensagens)**
Biddle2002b	**2,3**	**1**	**1**	**2**
Fayad2003	**1**	**1**	**1**	**2**

Todos os autores identificam a técnica dos Cartões CRC como um técnica simples e fácil de usar para introdução dos conceitos de orientação por objectos no ensino ou para iniciar as fases iniciais do desenvolvimento de software. No entanto, ao analisarmos a tabelas, podemos constatar que todas as abordagens propostas concentram os seus esforços nas fases iniciais do ciclo de vida do software e nenhuma nas fases tardias, isto porque, que seja do meu conhecimento, nenhum autor identificou esta técnica como sendo uma boa abordagem para a compreensão e recuperação de sistemas de legados. A verdade é que estes cartões permitem representar como os blocos arquitecturais (classes) colaboram entre si para implementar cada requisito de negócio. Esta visão apoia actividades como a manutenção do software ou a análise de impacto de alterações.

Por outro lado, a utilização da técnica manualmente pode tornar-se muito complexa quando estamos a falar de sistemas de grande dimensão, como muitos dos que são desenvolvidos actualmente. Embora, a maioria das propostas apresentadas siga esta abordagem, o apoio de ferramentas para a geração dos cartões é fundamental, principalmente no contexto de recuperação de sistemas legados.

Outro ponto a verificar é a origem dos dados para a criação dos cartões. Nas fases iniciais do ciclo de vida faz sentido que seja através de inspecção visual, ou até de técnicas mais complexas como o *parsing* de ficheiros, mas no contexto desta dissertação, não necessita de ser assim. Se a geração for efectuada tendo como ponto de partida a execução de um sistema, podemos garantir a sua fiabilidade dos resultados, uma vez que as informações recolhidas vão representar realmente o que está implementado.

6.8 Matrizes CRUD

Inicialmente as matrizes CRUD foram criadas para serem usadas no contexto de base de dados relacionais, no entanto, o uso destas matrizes tem evoluído e podemos vê-las actualmente a serem usadas em vários outros contextos. Elas são usadas para mapear interfaces com as possíveis interacções do utilizador [Wikipedia, 2008], mas também para estruturar processos de negócio [Wang, Xu et al., 2005], entre outros.

Estas matrizes são criadas normalmente na fase de análise e desenho, na identificação e estruturação das responsabilidades dos processos e dados, no entanto elas são bastante úteis noutras fases e com outras finalidades (ver capítulo 2)

Recentemente a técnica das matrizes CRUD começou a estender-se à análise e desenho OO (*Object-Oriented*) [Brandon, 2002], tendo sido propostas algumas abordagens possíveis para adoptar os conceitos OO nas matrizes. De acordo com os princípios do OO, cada classe deveria encapsular o seu estado interno, recorrendo a métodos para permitir acções como *create, read, update* e *delete* a cada membro de dados. No entanto, isto nada nos diz sobre a real utilização dos métodos ou sobre a sua adequada estruturação.

As matrizes CRUD OO podem ter vários níveis de granularidade. Normalmente, representam nas linhas as classes e nas colunas os casos de utilização, descritos nos diagramas de casos de utilização, ou os seus cenários. Existem outras abordagens que colocam nas linhas métodos, em vez das classes [Brandon, 2002].

A maneira como a matriz é gerada também é um ponto importante. Usualmente ela é criada manualmente, mas para sistemas grandes e complexos, isso pode ser uma tarefa árdua e até desmotivante. É evidente que um mecanismo de geração automática de tal matriz é útil neste tipo de situações, porém só possível na presença de um sistema construído (numa fase mais tardia do ciclo de vida do software).

Em [Brandon, 2002] é proposta uma abordagem de criação das matrizes CRUD baseadas em diagramas de actividades, que vão progressivamente sendo detalhados, até ao nível dos métodos. É seguida uma metodologia orientada às responsabilidades, que sugere a identificação das operações em primeiro lugar, para depois determinar os atributos de cada classe na estrutura do sistema. Para tal, os diagramas de actividades construídos vão incluir uma notação proposta de identificação única de cada operação. No final desta construção é criada manualmente uma matriz CRUD para cada cenário, descrevendo nas linhas cada classe e respectivos métodos, nas colunas os vários tipos de operações possíveis e nas células a identificação única dos métodos que realizam as operações nos diagramas de actividades. A esta abordagem de matrizes CRUD os autores deram o nome de “Detailed OO CRUD”.

Na minha óptica, esta abordagem não é muito vantajosa. O *overhead* da construção e detalhe dos diagramas de actividades, com a criação da matriz, para além de apresentar um consumo extra de tempo e esforço, já por si escasso no mundo empresarial, não gera resultados compensatórios. Não é possível ter uma visão global do sistema, a não ser que se junte as várias matrizes individuais, mas deste modo a matriz deixa de ser tão compacta e de tão fácil análise. Por outro lado, o detalhe dos diagramas de actividades até chegar ao nível do método, é desnecessário e

inadequado já que existem outros diagramas que melhor se adequam a esse efeito, como os diagramas de sequência.

Em linguagem orientadas a objectos, como o Java por exemplo, existe ainda o problema da definição do que é um *delete*. Na verdade é o sistema de *garbage collection* que "apaga" os objectos. Uma solução possível de resolver este problema, pode ser a identificação do âmbito dos objectos criados. Se um objecto é criado no âmbito de um método e não é retornado, pode-se dizer que no final do método o objecto é apagado porque não faz sentido fora do âmbito.

Apesar de terem sido feitas várias pesquisas sobre o uso de matrizes CRUD, principalmente no contexto de orientação a objectos, os resultados obtidos não forma muito satisfatórios. Na verdade, muito pouco foi publicado a este respeito, e portanto, não achei necessário criar uma taxionomia para orientar a análise deste assunto.

6.9 Testes de Cobertura e sua representação

A fase de testes está sempre presente no ciclo de vida de software, seja qual for a abordagem usada (modelo em cascata [Wikipedia, 2008], modelo em espiral [Wikipedia, 2007], RUP [Corporation, 2001], etc). Testar um sistema é muito importante para se conseguir produzir software fiável e obedecendo aos requisitos. Muitas vezes não é possível identificar, de forma viável, que partes do sistema são testadas ou sequer executadas. Por esta razão, várias ferramentas têm sido desenvolvidas ao longo dos últimos anos, na tentativa de suportar, não só o processo de testes, mas também a identificação das partes de um sistema que não estão a ser testadas.

De um modo geral, as ferramentas de testes podem ser divididas em quatro categorias diferentes:

- **Análise de modelos** - Um exemplo disso é a ferramenta *MetricView* [Chaudron and Lange, 2006]. Na *MetricView*, o objectivo é dar uma visão melhorada dos diagramas UML, permitindo visualizar métricas de software que foram computadas por outra ferramenta, directamente na representação gráfica dos modelos UML. A ideia é adicionar mecanismos gráficos para representar nos diagramas UML, de modo fácil e claro, os resultados da aplicação de métricas que estão normalmente sob a forma tabular (ver figura 7). Alguns dos mecanismos usados são gráficos, sinais de cruzes e vistos,

cores, etc. No entanto esta ferramenta é bastante limitada, na medida em que o foco de utilização é na forma de representação, ou seja, não produz nenhum resultado no âmbito dos teste, nem da aplicação das métricas em si.

Figura 114 Processo descrito em *MetricView*, em que a partir de um digrama UML e de resultados de métricas em formato tabular, se consegue um novo modo de visualização dos diagramas

- **Teste estático** – o teste estático pode englobar vários tipos de ferramentas, entre outros, ferramentas de análise de estrutura, de análise sintáxica e de análise de comportamento. A ferramenta Cantata++ [IPL, 2008] é um bom exemplo de uma ferramenta que, através análise estática, permite guiar o processo de testes de integração e unidade, oferecendo um conjunto de facilidades de teste, análise de cobertura e de análise de complexidade (exemplo: suporta as métricas MOOD [Abreu, 1993]).

- **Teste dinâmico** – o teste dinâmico é suportado por vários tipos de ferramentas, como ferramentas de monitorização, de testes de cobertura e *debuggers*. O artigo [Rountev, Kagan et al., 2004], descrito da secção das capturas de diagramas de sequência, propõe a implementação de uma ferramenta do tipo de análise de cobertura, face à análise do comportamento de um sistema. A ideia é usar a análise estática para definir os requisitos de cobertura para uma dada amostra e depois, durante a execução do sistema, analisar dinamicamente os resultados, de modo a identificar a percentagem da cobertura de acordo com os requisitos definidos. Os resultados são depois apresentados sob a forma de árvores de contexto de invocação.

- **Suporte aos testes** – este suporte pode abranger vários tipos de ferramentas, tais como ferramentas de desenho dos testes, ferramentas de planeamento, de gestão de defeitos e de gestão de requisitos. Um exemplo de uma ferramenta de gestão de requisitos é a *TBreq* [TBreq, 2008], que a partir da integração com a ferramenta *LDRA* (que inclui a *LDRA Testbed* e a *TBrun* para testar componentes), apresenta uma solução única que pode ajudar ao desafio de

mapear a especificação dos testes, os cenários de teste unitários, os dados dos testes e a verificação da cobertura do código e os requisitos de desenho. Ela permite assegurar a rastreabilidade ao longo do ciclo de vida do software e assegurar a completude da cobertura dos requisitos.

Embora existam várias abordagens propostas para a realização dos testes de cobertura, o modo de representação dos resultados obtidos é, na maior partes das vezes, tabular, gráfico ou texto estruturado (ex. árvores). Sendo os diagramas UML uma representação das várias vistas de um sistema, parece-me adequado representar os resultados da cobertura directamente sobre os elementos dos diagramas. Se, por exemplo, estivermos a analisar a cobertura de classes, parece-me lógico identificar no diagrama de classes do sistema, quais as que foram cobertas pela execução ou teste do mesmo. As maneiras de identificar essa informação podem ser muito variadas, algumas até descritas em [Chaudron and Lange, 2006]. No entanto, a mais simples e, de certo modo, mais adequada seria a utilização de uma palete de cores, indicativa da percentagem de cobertura capturada.

Na verdade, as cores são muitas usadas para variados fins. Em [Malloy and Power, 2005], por exemplo, a cor é usada nos diagramas de sequência e de colaboração para indicar os objectos que são instância da mesma classe. Também em [Lange and Chaudron, 2005] é proposto usar técnicas comummente vistas em sistemas de informação geográficos, como cores, pontos, linhas, setas, gráficos e densidades de pontos, para representar os resultados da aplicação de métricas aos elementos usados na modelação com UML.

Não existe, tanto quanto sei, uma abordagem que descreva de forma combinada, o modo de realizar os testes de coberturas ou de análise, em junção com os mecanismos de visualização.

Capítulo 7

Conclusões e trabalho futuro

Conteúdo

Neste capítulo são apresentadas as conclusões para esta dissertação, identificando as linhas de orientação gerais para a continuação futura do trabalho desenvolvido.

7 Conclusões e trabalho futuro

7.1 Introdução Geral

Este último capítulo revê as contribuições presentes na introdução e descreve brevemente a forma como o trabalho foi desenvolvido para as tentar atingir. Por fim, é feita uma apreciação do trabalho futuro que surgiu na realização do trabalho desta dissertação.

7.2 Conclusões

Ao longo dos anos tem havido uma grande evolução no desenvolvimento dos sistemas de *software*, que apresentam muitas vezes requisitos difíceis de cumprir, quer a nível de prazos, quer de orçamento. Estes requisitos são por vezes tão esmagadores que acabam por trazer problemas ao próprio desenvolvimento e manutenção dos sistemas, que acabam por não ser devidamente documentados. Neste contexto, o desenvolvimento apoiado por ferramentas e abstracções é essencial para se conseguir atingir software de qualidade. Contudo a velocidade a que hoje se processa a informação é tal, que estas ferramentas e abstracções não podem significar um consumo de tempo para quem os utiliza, pois quando assim é, estas são muito facilmente descartadas. No entanto, sistemas sem a documentação apropriada são um real problema para as empresas e limitam a sua extensão, alteração, manutenção e gestão.

Nesta dissertação foi proposto um novo mecanismo de geração automática de artefactos a partir de sistemas existentes, de uma forma e num contexto inovador. O mecanismo inicia-se com uma captura dinâmica de um sistema, onde são recolhidos dados para posteriormente representar quer a visão estrutural, quer a visão comportamental do mesmo. Isto permite garantir a rastreabilidade, através da ligação que fica criada desde os modelos ao código fonte.

De uma forma geral esta dissertação procurou guiar o processo de recuperação e entendimento de sistemas legados, minimizando o tempo e o esforço habitualmente necessário para o conseguir. Dos objectivos previstos inicialmente, praticamente todos foram atingidos, graças aos artefactos produzidos.

Os cartões CRC estendidos e as matrizes CRUD tornaram possível apoiar as actividades de análise de impacto e manutenção. Estes artefactos representam como as classes do sistema estão implementadas e como estas se relacionam entre si, ficando visível como os requisitos estão implementados. Quando é necessário efectuar uma alteração em alguma parte do sistema, é possível ter uma melhor percepção de que outras áreas necessitam de ser também alteradas.

Na análise da qualidade e completude do modelo de dados, face aos requisitos que o sistema deve suportar, é comum representar-se os resultados sob a forma de uma matriz CRUD. Ela permite encontrar falhas no modelo de dados e adequar os dados necessários. Por outro lado, a sua estrutura permite analisar que classes ou entidades são usadas, e de que forma, facilitando a identificação de fontes de problemas de performance, como *bottlenecks* [Answers, 2001]. A sua forma de compactação e representação das estruturas e suas ligações pode ainda ajudar a guiar a construção de baterias de testes para sistemas de grande dimensão.

A actividade de testes também ficou beneficiada com os artefactos produzidos. Os diagramas coloridos representam de uma forma simples e diagramática a cobertura, quer das capturas dos cenários do sistema, quer da cobertura de execução da estrutura interna do mesmo.

Foi ainda possível diminuir o fosso existente entre a modelação estática e dinâmica, que leva a perdas de rastreabilidade. Este é o caso do hiato do UML no desenvolvimento dos diagramas de casos de utilização e dos diagramas de classes, em que depois não existe a ligação entre ambos, representado pelo diagrama de sequência. Nesta dissertação, essa ligação é gerada automaticamente sob a forma de diagramas de sequência temporizados. Para além disso, também os diagramas coloridos gerados no âmbito dos testes de cobertura são um bom exemplo da união dos dois tipos de modelação, uma vez que para os criar é feita uma comparação do que foi definido de forma estática, como um diagrama de classes, com o que foi recolhido na captura do comportamento do sistema.

Outro factor relevante da dissertação foi o modo de criação dos artefactos. A geração automática veio reduzir grandemente a dificuldade em conseguir os artefactos documentais do sistema, que leva muitas empresas a deixar estas actividades para segundo plano, originando novos sistemas legados. O modo de geração apresentado vem reduzir os custos de tempo e esforço habitualmente necessários para os produzir. Para além disso, as novas características propostas em

cada artefacto, face aos originais, aumentam a simplicidade de análise e compreensão dos mesmos.

7.3 Evolução futura

7.3.1 Melhorar o desempenho

Actualmente o *ReModeler* é um sistema completo, constituído por vários componentes, podendo ser destacados uma interface gráfica, componentes de geração da documentação e componentes para capturar a execução do sistema. Todos estes componentes são entrelaçados com o sistema em análise e ambos executam na mesma máquina virtual (JVM). As funcionalidades e o volume de dados com os quais o *ReModeler* tem de lidar são bastante exigentes a nível computacional. Se o sistema em estudo também necessitar de muito poder computacional, o desempenho de ambos pode baixar. Uma maneira para tentar mitigar este problema é desacoplar as várias funcionalidades do *ReModeler* em componentes independentes, criando um conjunto de aplicações que cooperam entre si. A separação dos conceitos em componentes independentes (ou bibliotecas) não só permite melhorar o desempenho, mas também tornará o sistema mais leve e fácil de usar, uma vez que permitirá o uso de cada componente consoante a necessidade do utilizador. A comunicação entre os vários componentes é assegurada pela base de dados que já representa, na implementação actual, um meio de ligação entre as várias funcionalidades. No diagrama de componentes apresentado no capítulo 3, é possível observar que o sistema *ReModeler* funciona como uma aplicação única, no entanto, a proposta da apresentada na Figura 115 vai no sentido de criar aplicações independentes que comunicam entre si para realizarem as funcionalidades já existentes. Por exemplo, a funcionalidade de capturar a execução do sistema em análise ficaria a cargo do componente *ReModeler Scenario Capturer*, que se comportaria como uma aplicação independente só activada por opção do utilizador. Apenas este componente passaria a ter de estar entrelaçado com o sistema em análise, o que significaria uma redução da complexidade e intrusão, aumentando o desempenho. A geração dos diagramas de sequência temporizados ficaria a cargo do componente *Timed Sequence Diagrams & Static Component* que apenas teria de ler os dados armazenados na base de dados do *ReModeler* recolhidos pelo componente das capturas.

Mesmo com este desacoplar de componentes, pode ainda ser interessante incluir um mecanismo de monitorização da utilização da memória e da performance, podendo mesmo ser necessário fazer alguma alteração estrutural para optimizar a desempenho geral.

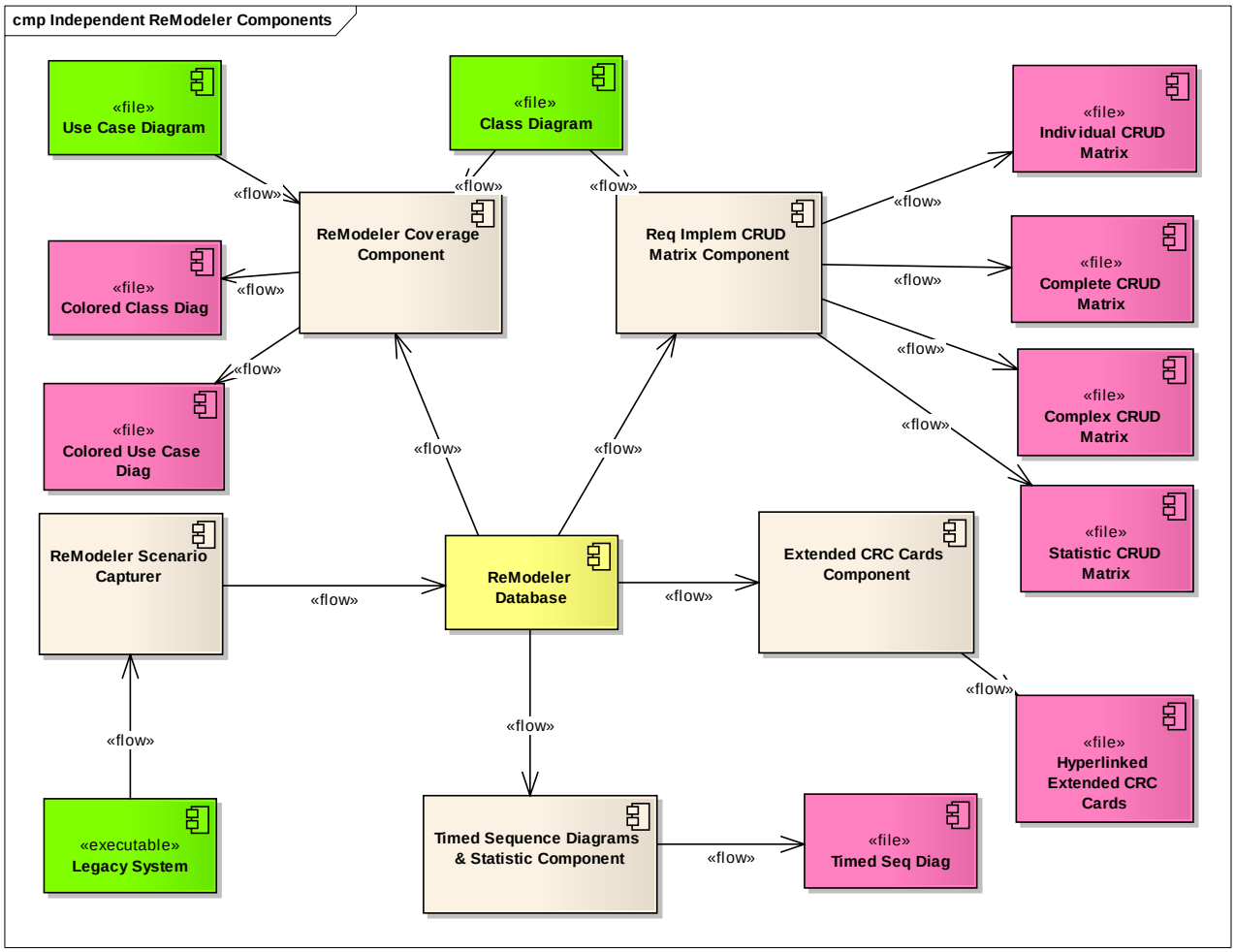


Figura 115. Diagrama de Componentes com independência.

7.3.2 Estender a outras linguagens

O *ReModeler* foi implementado para suportar a documentação de sistemas escritos em Java. Isto poderá representar uma limitação face aos restantes sistemas que existem. A extensão da utilização a sistemas escritos noutras linguagens, como por exemplo o C++, pode ser uma mais-valia importante.

7.3.3 Captura e Geração de Diagramas de Sequência

A captura dinâmica do sistema foi implementada tirando partido do *weaving* da tecnologia dos aspectos. Estando na posse de parte ou da totalidade do sistema a analisar, em código fonte ou binário, é possível entrelaçar o sistema com o *ReModeler* e efectuar uma captura. Por cada execução do sistema vão sendo recolhidas as mensagens, e respectiva marcação temporal, que correspondem às invocações de métodos entre objectos. Estes dados são depois armazenados na base de dados do *ReModeler* e são posteriormente usados para gerar um conjunto de artefactos que permitem a compreensão, manutenção e teste do sistema. Um desses artefactos gerados é o diagrama de sequência temporizado, uma extensão do existente em UML. Esse diagrama é exportado para um ficheiro em formato XMI, para permitir a visualização numa ferramenta de UML externa. De modo a permitir a escolha da informação relevante a analisar, é dada a possibilidade ao utilizador de usar um mecanismo de filtragem, que também ele fica guardado na base de dados, para futuras utilizações. O quadro da Figura 116 representa esquematicamente o resultado da implementação desta captura e consequente geração do diagrama, usando os mesmos critérios definidos na revisão do estado da arte.

Critérios	Mecanismo de instrumentação	Modo de Filtragem	Tipo de Captura	Alvo de Captura	Abrangência	Armazenamento e representação dos dados	Exportação	Modo de visualização	Finalidade	Granularidade
ReModeler 2008 - Capture & Timed Seq Diag	**4**	**4**	**3**	**1,2**	**1**	**c,c**	**5**	**2**	**1,2,3**	**3**

Figura 116. Quadro de classificação dos componentes de capturas e geração de diagramas de sequência face à taxionomia apresentada.

Actualmente, muitos dos sistemas desenvolvidos usam facilidades de paralelismo (*multithreading*). No entanto, a implementação da captura não prevê este tipo de mensagens. Futuramente considero que seja importante adicionar esta particularidade, refinando os aspectos já implementados, de modo a melhorar a captura já realizada e de modo a incluir a captura deste tipo de eventos.

Para além disso, devido ao consumo de tempo e recursos que pode ter uma captura, este deveria ser o componente mais independente dos restantes constituintes do *ReModeler*, criando como foi referido atrás uma aplicação independente que apenas capturasse e guardasse os dados na base de dados de forma persistente.

Outro ponto que requer alguns melhoramentos é o sistema de filtragem. É especialmente vantajoso incluir nas opções de filtragem informação estatística das invocações, de modo a melhor guiar o processo do utilizador.

Faz também sentido conseguir identificar, através da inclusão de informação no XMI, as partes de cada diagrama de sequência que correspondem a cada passo (oriundos da descrição dos cenários), como por exemplo apresentar cores diferentes dos elementos sempre que se muda de passo.

7.3.4 Cartões CRC

Também os cartões CRC estendidos são gerados automaticamente a partir das informações que foram recolhidas da captura do sistema e que foram posteriormente armazenadas na base de dados. Estes cartões representam, para cada classe, quais os cenários em que ela participa e quais as outras classes que a auxiliam a implementar cada cenário. No quadro da Figura 117 são apresentados os resultados da análise dos cartões CRC propostos face aos critérios presentes na revisão do estado da arte.

Este artefacto também tem alguns melhoramentos previstos para realização futura. Neste momento os cartões são gerados em formato HTML estático. Deixo para trabalho futuro a adaptação da construção actual para uma construção dinâmica usando *JavaServer Pages (JSPs)* [Sun Microsystems, 2008] que lêem os dados directamente da base de dados e que conseguem, por exemplo, mostrar de forma ligada as descrições de cada cenário ou outras informações relevantes.

Critérios	Fase de desenvolvimento	Geração	Origem	Granularidade
ReModeler2008 - Extended CRC Cards	**4**	**4**	**3**	**2**

Figura 117. Quadro de classificação do componente de geração dos cartões CRC face à taxionomia apresentada.

7.3.5 Matrizes CRUD

As matrizes propostas nesta dissertação são geradas nas fases mais tardias do ciclo de vida do software, quando já existe pelo menos parte do sistema desenvolvido. Isto permite a captura de informações provenientes da execução do sistema em análise, permitindo a geração automática das matrizes, o que garante um elevado grau de fiabilidade. A granularidade destas matrizes é variável, sendo dada a possibilidade ao utilizador de escolher a visão que deseja analisar, mais ou menos detalhada.

As matrizes CRUD também foram implementadas de modo estático, recorrendo à tecnologia de HTML. Tal como acontece para os cartões CRC, a adaptação da implementação actual para uma mais dinâmica, tirando partido da tecnologia de JSPs, é deixada para trabalho futuro. Este modo iria permitir a granularidade fosse escolhida, através de uma estrutura em árvore, à medida que se processava a actividade de análise da matriz, podendo diminuir ou aumentar.

7.3.6 Relatórios Estatísticos

A análise de cobertura tem uma representação qualitativa, mas num processo de desenvolvimento mais controlado é necessária informação quantitativa, tal como a que podemos encontrar descrita dos níveis mais altos de *frameworks* de processos como o CMMI ou a ISO 15504 (SPICE). Prevejo a criação de relatórios que apresentem não só as percentagens de cobertura, já presentes sob o formato de diagramas coloridos, e utilização média de recursos (CPU e memória), mas também um conjunto de estatísticas e métricas do processo de captura do cenário em diferentes níveis de abstracção.

7.3.7 Estimação de Custos

Embora alguns dos artefactos gerados permitam ter uma ideia das dependências geradas pela alteração de um determinado requisito, como as matrizes CRUD e os cartões CRC, não existe um real planeamento de esforço/custos, para o conseguir. Penso que seria interessante analisar as informações já existentes, de modo a criar um grafo de dependências (incluindo as indirectas) que permita criar um modelo de estimação. Este modelo de estimação permitiria estimar o esforço,

bem como os custos, necessários para a concretização de alterações, que apoiariam as decisões dos responsáveis pela gestão do projecto e pela gestão de alterações.

7.3.8 Diagramas coloridos

Os diagramas propostos nesta dissertação mostram a cobertura de capturas de cenários do sistema, usando para isso os diagramas de casos de utilização coloridos e mostram a cobertura da estrutura na execução do sistema, através dos diagramas de classes coloridos. No entanto, pode ser interessante ver a análise de cobertura a outros níveis de abstracção. Os intervenientes podem querer saber, para além da cobertura da captura, quais os casos de utilização que já estão documentados, ou seja, quais os casos de utilização que têm os seus cenários devidamente descritos. Para demonstrar isso, é possível criar um diagrama de casos de utilização que, também através de uma escala de cores, indique a cobertura da descrição dos cenários.

Por outro lado, os programadores e engenheiros de teste podem querer identificar que partes do sistema estão a ser usadas no contexto de um dado cenário, caso de utilização ou bateria de testes. Em parte, tal já é possível, através do diagrama de classes colorido, mas se o sistema em análise for muito grande ou complexo, eles podem necessitar de ver a cobertura de execução do sistema com uma granularidade ao nível do pacote. Para isso, é interessante criar futuramente diagramas de pacotes coloridos que mostrem a cobertura de execução, de acordo com a percentagem das classes que o constituem e que foram executadas

Anexo A

Tecnologias Usadas na Implementação do *ReModeler*

Conteúdo

A. Tecnologias Usadas na Implementação do *ReModeler*

a. *Java Database Connectivity Application Program Interface*

Na implementação da aplicação do *ReModeler* a comunicação com a base de dados foi realizada através da interface de programação *Java Database Connectivity Application Program Interface* (JDBC API) [Sun Microsystems, 2008], que é um caso particular da norma *Open Database Connectivity* (ODBC) [Wikipedia, 2008] para a conexão entre a linguagem de programação Java e um grande número de SGBDs SQL. A *Open Database Connectivity* (ODBC) disponibiliza uma interface de programação padronizada para que aplicações cliente possam interagir com SGBDs. A norma ODBC foi criada com o intuito de ser independente de linguagens de programação, SGBDs e sistemas operativos.

b. *AspectJ*

Aspect-oriented programming (AOP) [Wikipedia, 2008] é um paradigma de programação recente que facilita a modularização dos *concerns* no desenvolvimento de software. Um concern é uma parte de interesse de um programa, normalmente associado a funcionalidades ou comportamentos por ele disponibilizados.

Em particular, na AOP são extraídos os *concerns* espalhados pelas classes que são tratados de forma transversal e unificada numa única estrutura, o aspecto. O desacoplamento destes *concerns* das classes e o seu posicionamento em aspectos alivia as classes originais de gerir funcionalidades ortogonais. Mais tarde, o código do aspecto é injectado nos sítios apropriados, dentro das classes, por um processo chamado de *weaving*. Os aspectos contêm *join points* que especificam pontos de execução bem definidos do programa a instrumentar, como a invocação ou execução de um método específico. Os *poincuts* descrevem conjuntos de *joint points* especificando, por exemplo, os objectos e métodos a considerar. Um *advice* é o código adicional que deve ser executado antes (*before*) ou depois (*after*) dos *join points*. Ele pode ainda ter o controlo de quando um *join point* pode correr ou não.

Os sistemas analisados nesta dissertação são sistemas escritos em linguagem Java. Por esta razão, foi usado o *AspectJ*, que é uma conhecida implementação de AOP para Java. O *AspectJ* está estruturado segundo uma sintaxe bem definida [PaloAlto, 2003] (ver Figura 118). A primeira linha de um aspecto especifica a sua assinatura, identificando um dos três tipos possíveis de *advices*: *before* (usado para executar código antes do *pointcut*), *after* (usado para executar código mesmo depois do *pointcut*) e *around* (usado para executar código antes e depois do *pointcut*). De seguida é feita a declaração do *pointcut* onde estão disponíveis diferentes funcionalidades. Por exemplo, é possível, usando as funcionalidades de *call* e *within*, especificar um *pointcut* como qualquer invocação de um método com nome específico, de uma classe e pacote específicos. Depois da declaração do *pointcut* são descritas as acções do próprio *advice*, onde se diz o que vai ser feito antes, depois ou em vez da execução do *pointcut*. Quando se usa esta última opção é possível alterar completamente o comportamento do sistema original, ou então executar o *pointcut* original usando o método *proceed()* do *AspectJ*.

```
around(): execution(* * *.getName()) && !within(* * somePackage)
{
// any Java statement
proceed();
// any Java statement
}
```

Figura 118. Estrutura de um aspecto em *AspectJ* (retirado de [Briand, Labiche et al., 2003]).

c. *SAX*

O processamento de XML em linguagem Java pode ser feito de várias maneiras. Actualmente existem várias APIs que permitem escrever aplicações para o efeito, sendo que, de entre as mais conhecidas, se podem destacar o SAX e o DOM [Developerlife.com, 2008]. Ambas as APIs foram desenhadas para permitir o acesso dos programadores às informações contidas em ficheiros XML, sem terem de recorrer a um interpretador da linguagem de programação. No entanto, a abordagem usada para o fazer varia bastante de um para o outro. O DOM permite o acesso à informação através de uma estrutura hierárquica. Ele cria uma árvore de nós (baseada na estrutura do documento XML) em memória transiente, para onde carregar toda a informação contida no documento. No caso do SAX, ele permite o acesso à

informação através de sequências de eventos. No processo do *ReModeler*, os ficheiros XML que necessitam de ser lidos podem apresentar tamanhos bastante grandes, o que faz da possibilidade de carregar toda a informação, uma solução fraca a nível de desempenho. Por esta razão a API escolhida para a implementação do *ReModeler* foi o SAX.

SAX [Network World, 2008] é uma interface comum implementada para diferentes *parsers* XML, baseada em eventos, operando sob o princípio do *callback*. Uma aplicação cria um objecto SAX *parser* e passa-lhe como elementos de entrada um ficheiro XML e um *document handler*, que recebe *callbacks* para os eventos SAX. O *parser* SAX converte os *inputs* numa *stream* de eventos correspondentes a elementos estruturais do *input*, como *tags* de XML ou blocos de texto. Quando um evento ocorre, ele é passado ao método apropriado, definido pelo programador, que implementa a interface *callback org.xml.sax.DocumentHandler*.

Anexo B

Ferramentas de Reverse Engineering

Conteúdo

B. Ferramentas de Reverse Engineering

a. Descrição de Ferramentas de Reverse Engineering

O processo proposto nesta dissertação pode ser classificado como fazendo parte do processo de *Reverse Engineering*. Actualmente existem inúmeras ferramentas no mercado com facilidades de *reverse engineering* que permitem recolher artefactos documentais para sistemas em análise. No entanto, nem todos os artefactos propostos nesta dissertação são conseguidos e alguns dos que são, apresentam falhas, quer ao nível do processo de obtenção, quer ao nível dos resultados obtidos.

O primeiro passo descrito no processo do *ReModeler* é a documentação das funcionalidades do sistema em análise, os seus casos de utilização e descrição dos respectivos cenários. Actualmente existem no mercado, ferramentas que permitem criar e documentar os casos de utilização para um sistema, como o *Visual Use Case 2006* [TechnoSolutions, 2006] e ferramentas UML para criar os diagramas de casos de utilização. No entanto, não é frequente encontrar ferramentas que adicionem as funcionalidades de criar diagramas de casos de utilização com a notação da UML, em conjunto com a possibilidade de documentar os cenários, desses casos de utilização, de forma estruturada.

No *ReModeler*, a geração de artefactos inicia-se com a captura de execução de cada cenário de um caso de utilização. Um dos artefactos gerados desta captura é o diagrama de sequência UML, cujos dados são obtidos de forma dinâmica extraída da execução do próprio sistema. Existem ferramentas disponíveis no mercado que dizem gerar diagramas de sequência de forma automática, a partir de texto [Strauch, 2008], a partir de listas de eventos [VisualParadigm, 2007], a partir da análise do sistema de forma estática (Flowchart4j [CodeSWAT, 2008]) ou dinâmica ([MaintainJ, 2008] ou [Dmitry, 2008]) (vide capítulo de trabalho relacionado).

A criação de cartões CRC, numa fase inicial do ciclo de vida de software, está bem documentada em alguns dos documentos apresentados no capítulo do trabalho relacionado. Actualmente existem algumas ferramentas que acompanham a criação computorizada dos cartões [VisualParadigm, 2007]. Que seja do meu conhecimento não existem no entanto ferramentas que efectuem o *reverse engineering* dos mesmos.

As matrizes CRUD foram inicialmente propostas para base de dados relacionais. Nesse contexto existem algumas ferramentas que produzem essas

matrizes, algumas delas de um modo quase automatizado. Quando a procura de ferramentas é feita para matrizes CRUD em contextos orientados a objectos, os resultados obtidos são inexistentes.

No caso dos testes, de cobertura e intensidade de utilização, existem inúmeras ferramentas que produzem este tipo de testes, mas normalmente usam outros mecanismos de apresentação dos resultados, que não os diagramas UML coloridos (vide capítulo Trabalho Relacionado).

Bibliografia

Abreu, F. B. (1993). Metrics for Object Oriented Software Development. 3rd International Conference on Software Quality, Lake Tahoe, Nevada, EUA, American Society for Quality.

Abreu, F. B. e., F. Silva, et al. (2007). Model-Driven Testing for Java Projects. expo:QA'07. Madrid.

Altova. (2008). "XML Editor, Data Management, UML and Web Services tools." 2008, from http://www.altova.com/.

Ambler, S. W. (1998). CRC Modeling: Bridging the Communication Gap Between Developers and Users, AmbySoft Inc.

Answers, D. (2001). CRUD Matrix. http://databaseanswers.org/data_migration/crud_matrix.htm, accessed on January 15.

Beck, K. and W. Cunningham (1989). A laboratory for teaching object oriented thinking. Conference proceedings on Object-oriented programming systems, languages and applications. New Orleans, Louisiana, United States, ACM.

Biddle, R., J. Noble, et al. (2002). "Essential use cases and responsibility in object-oriented development." Aust. Comput. Sci. Commun. **24**(1): 7-16.

Biddle, R., J. Noble, et al. (2002). Reflections on CRC cards and OO design. Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications. Sydney, Australia, Australian Computer Society, Inc.

Borstler, J. (2005). Improving CRC-card role-play with role-play diagrams. Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. San Diego, CA, USA, ACM.

Brandon, D. J. (2002). "Crud matrices for detailed object oriented design." J. Comput. Small Coll. **18**(2): 306-322.

Briand, L. C., Y. Labiche, et al. (2003). Towards the Reverse Engineering of UML Sequence Diagrams. Proceedings of the 10th Working Conference on Reverse Engineering, IEEE Computer Society.

Chaudron, M. and C. Lange (2006). MetricView. Eindhoven.

CMUniversity, C. M. U. (2008). "What is CMMI?" Retrieved January 10 2008, from http://www.sei.cmu.edu/cmmi/general/.

CodeSWAT (2008). Flowchart4j.

Corporation, R. S. (2001). Rational Unified Process: Visão Geral. http://www.wthreex.com/rup/, accessed on January 15.

Delgado, S. (2006). Next-Generation Techniques for Tracking Design Requirements Coverage in Automatic Test Software Development. IEEE.

Developerlife.com. (2008). "Should I use SAX or DOM?", 2008, from http://developerlife.com/tutorials/?p=28.

Dmitry, B. (2008). jtracert: Automatically generate sequence UML diagrams directly from your java code runtime.

Eclipse, F. (2008). " Aspectj: Crosscutting objects for better modularity." Retrieved 10-07-08, 2008, from http://www.eclipse.org/aspectj/

Eclipse, F. (2008). "Eclipse Foundation." 2008, from http://www.eclipse.org/.

eTeks. (2006-2008). "Sweet Home 3D." 2008, from http://sweethome3d.sourceforge.net/.

Fayad, M. E., H. Hamza, et al. (2003). A pattern for an effective class responsibility collaborator (CRC) cards. Information Reuse and Integration, 2003. IRI 2003. IEEE International Conference on.

Gates, B. (2001). The PC: 20 Years Young. C. Microsoft. http://www.microsoft.com/presspass/ofnote/08-12pc20.mspx.

Gibbs, W. W. (1994). "Software's Chronic Crisis." Scientific American magazine.

Gouveia, V. (2008). Cenários Visuais: Rastreio de requisitos, Documentação e Animação para Sistemas Legados Departamento de Informática, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia.

Grünbacher, P. and Y. Ledru. (2004). "Automated Software Engineering — Introduction." 2008, from http://www.ercim.org/publication/Ercim_News/enw58/intro.html.

Hvam, L., R. Jesper, et al. (2003). "CRC cards for product modelling." Comput. Ind. **50**(1): 57-70.

IPL, S. P. G. (2008). Cantata++. Bath, UK.

Kilov, H. (1990). From semantic to object-oriented data modeling. Systems Integration, 1990. Systems Integration '90., Proceedings of the First International Conference on.

Kitchens, T. (2006). "Automating Software Development Processes." Retrieved 25-06-08, 2008, from http://www.developerdotstar.com/mag/articles/automate_software_process.html.

Lange, C. and M. Chaudron (2005). Combining metrics data and the structure of UML models using GIS visualization approaches. Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on.

MagicDraw, i. (2008). "Magicdraw: Architecture Made Simple." 2008, from http://www.magicdraw.com/.

MaintainJ, I. (2008). "MaintainJ: Reverse Engineer Java Like Never Before ".

Malloy, B. A. and J. F. Power (2005). Exploiting UML dynamic object modeling for the visualization of C++ programs. Proceedings of the 2005 ACM symposium on Software visualization. St. Louis, Missouri, ACM.

Merdes, M. and D. Dorsch (2006). Experiences with the development of a reverse engineering tool for UML sequence diagrams: a case study in modern Java development. Proceedings of the 4th international symposium on Principles and practice of programming in Java. Mannheim, Germany, ACM.

Network World, I. (2008). "Programming XML in Java, Part 1." 2008, from http://www.javaworld.com/jw-03-2000/jw-03-xmlsax.html.

Nunes, M. and H. O'Neill (2004). Fundamental de UML, FCA.

Objecteering, S. (2008). "Objecteering: The model-driven development tool." 2008, from http://www.objecteering.com/.

OMG (2007). Unified Modeling Language: Superstructure - version 2.1.1**:** 732.

OMG. (2008). "MOF 2.0 / XMI Mapping Specification, v2.1.1." Retrieved 2008, from http://www.omg.org/technology/documents/formal/xmi.htm.

PaloAlto, R. C. (2003). "The AspectJ Programming Guide." 2008, from http://www.eclipse.org/aspectj/doc/released/progguide/index.html.

PostgreSQL, G. D. G. "PostgreSQL: The World's most advanced open source database." Retrieved 11-07-2008, 2008, from http://www.postgresql.org/

Pressman, R. S. (2005). Software Engineering: A Practitioner's Approach, McGraw-Hill Book Company.

Quilici, A. (1995). Reverse engineering of legacy systems: a path toward success. Proceedings of the 17th international conference on Software engineering. Seattle, Washington, United States, ACM.

Raman, A. and S. Tyszberowicz (2007). The EasyCRC Tool. Software Engineering Advances, 2007. ICSEA 2007. International Conference on.

Rational (2001). Rational Unified Process: Best Practices for Software Development Teams, Rational Software.

Rational, S. C. (2001). "Conceitos: Principais Medidas de Teste." 2008, from http://www.wthreex.com/rup/process/workflow/test/co_keyme.htm.

Rountev, A., S. Kagan, et al. (2004). Static and dynamic analysis of call chains in java. Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. Boston, Massachusetts, USA, ACM.

Rountev, A., O. Volgin, et al. (2005). Static control-flow analysis for reverse engineering of UML sequence diagrams. Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. Lisbon, Portugal, ACM.

Silva, A. and C. Videira (2005). UML, Metodologias e Ferramentas CASE, Centro Atlântico.

SourceForge. (2008). "SourceForge.Net: Open Source Software ", 2008, from http://sourceforge.net/.

SparxSystems. (2008). "Enterprise Architect." 2008, from http://www.sparxsystems.com/products/ea/index.html.

Strauch, M. (2008). Quick Sequence Diagram Editor

Sun Microsystems, I. (2008). "Java SE Technologies - Database." 2008, from http://java.sun.com/javase/technologies/database/.

Sun Microsystems, I. (2008). "JavaServer Pages Technology." 2008, from http://java.sun.com/products/jsp/.

Taniguchi, K., T. Ishio, et al. (2005). Extracting sequence diagram from execution trace of Java program. Principles of Software Evolution, Eighth International Workshop on.

TBreq. (2008). "LDRA: Requirements Traceability with TBreq." Retrieved January 30, 2008, from www.ldra.co.uk/tbreq.asp.

TechnoSolutions (2006). Visual Use Case 2006.

VisualParadigm (2007). VisualParadigm for UML.

Wang, Z., X. Xu, et al. (2005). "A Survey of Business Component Identification Methods and Related Techniques." INTERNATIONAL JOURNAL OF INFORMATION TECHNOLOGY **2**(4).

Wikipedia. (2007). "Spiral model." Retrieved January 15, 2008, from http://en.wikipedia.org/wiki/Spiral_model.

Wikipedia. (2008). "Aspect-oriented programming." 2008, from http://en.wikipedia.org/wiki/Aspect-oriented_programming.

Wikipedia. (2008). "Create, read, update and delete." Retrieved January 20
2008, from http://en.wikipedia.org/wiki/Create%2C_read%2C_update_and_delete.

Wikipedia. (2008). "Microsoft." Retrieved January 30
2008, from http://en.wikipedia.org/wiki/Microsoft#_note-1.

Wikipedia. (2008). "Open Database Connectivity." 2008, from http://en.wikipedia.org/wiki/ODBC.

Wikipedia. (2008). "Waterfall model." Retrieved January 15
2008, from http://en.wikipedia.org/wiki/Waterfall_model.